CHAPTER

Service-Oriented Architectures for Distributed Computing 5

CHAPTER OUTLINE

SUMMARY

In this chapter, we cover two major service-oriented architecture styles, namely REST (*REpresentational State Transfer*) and WS (*Web Services*) and their extensions. We discuss message-oriented middleware and enterprise bus infrastructure with a publish-subscribe infrastructure. Two application interfaces (OGCE and HUBzero) are described using web service (portlet) and Web 2.0 (gadget) technologies. We handle data and metadata in distributed systems using service registries and semantic web/grid. Finally, we describe a general workflow approach illustrating it with the BPEL web service standard, Pegasus, Taverna, Kepler, Trident, and Swift.

5.1 SERVICES AND SERVICE-ORIENTED ARCHITECTURE

Technology has advanced at breakneck speeds over the past decade, and many changes are still occurring. However, in this chaos, the value of building systems in terms of services has grown in acceptance and it has become a core idea of most distributed systems. Loose coupling and support of heterogeneous implementations makes services more attractive than distributed objects. Web services move beyond helping various types of applications to exchanging information. The technology also plays an increasingly important role in accessing, programming on, and integrating a set of new and existing applications.

We have introduced *service-oriented architecture* (SOA) in Section 1.4.1. In general, SOA is about how to design a software system that makes use of services of new or legacy applications through their published or discoverable interfaces. These applications are often distributed over the networks. SOA also aims to make service interoperability extensible and effective. It prompts architecture styles such as loose coupling, published interfaces, and a standard communication model in order to support this goal. The *World Wide Web Consortium* (W3C) defines SOA as a form of distributed systems architecture characterized by the following properties [1]:

Logical view: The SOA is an abstracted, logical view of actual programs, databases, business processes, and so on, defined in terms of what it does, typically carrying out a business-level operation. The service is formally defined in terms of the messages exchanged between provider agents and requester agents.

Message orientation: The internal structure of providers and requesters include the implementation language, process structure, and even database structure. These features are deliberately abstracted away in the SOA: Using the SOA discipline one does not and should not need to know how an agent implementing a service is constructed. A key benefit of this concerns legacy systems. By avoiding any knowledge of the internal structure of an agent, one can incorporate any software component or application to adhere to the formal service definition.

Description orientation: A service is described by machine-executable metadata. The description supports the public nature of the SOA: Only those details that are exposed to the public and are important for the use of the service should be included in the description. The semantics of a service should be documented, either directly or indirectly, by its description.

- **Granularity** Services tend to use a small number of operations with relatively large and complex messages.

- **Network orientation** Services tend to be oriented toward use over a network, though this is not an absolute requirement.
- **Platform-neutral** Messages are sent in a platform-neutral, standardized format delivered through the interfaces. XML is the most obvious format that meets this constraint.

Unlike the component-based model, which is based on design and development of tightly coupled components for processes within an enterprise, using different protocols and technologies such as CORBA and DCOM, SOA focuses on loosely coupled software applications running across different administrative domains, based on common protocols and technologies, such as HTTP and XML. SOA is related to early efforts on the architecture style of large-scale distributed systems, particularly *Representational State Transfer* (REST). Nowadays, REST still provides an alternative to the complex standard-driven web services technology and is used in many Web 2.0 services. In the following subsections, we introduce REST and standard-based SOA in distributed systems.

5.1.1 REST and Systems of Systems

REST is a software architecture style for distributed systems, particularly distributed hypermedia systems, such as the World Wide Web. It has recently gained popularity among enterprises such as Google, Amazon, Yahoo!, and especially social networks such as Facebook and Twitter because of its simplicity, and its ease of being published and consumed by clients. REST, shown in Figure 5.1, was introduced and explained by Roy Thomas Fielding, one of the principal authors of the HTTP specification, in his doctoral dissertation [2] in 2000 and was developed in parallel with the HTTP/1.1 protocol. The REST architectural style is based on four principles:

Resource Identification through URIs: The RESTful web service exposes a set of resources which identify targets of interaction with its clients. The key abstraction of information in REST

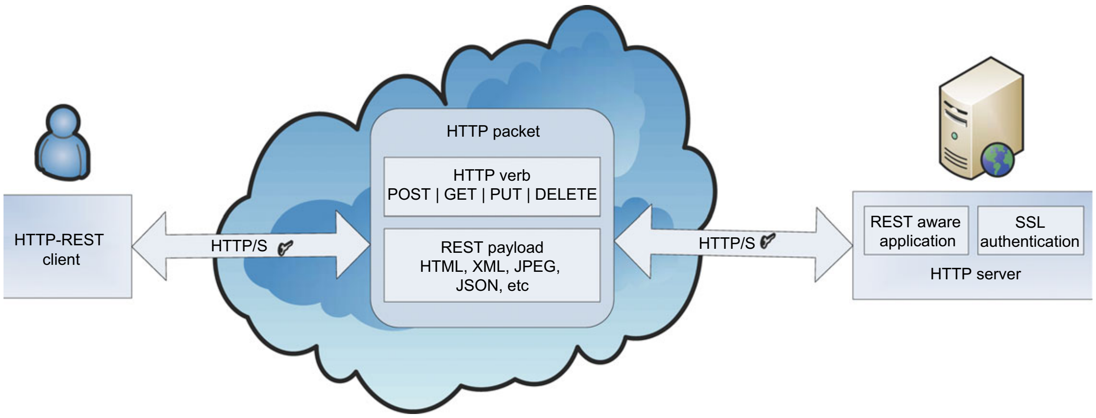

FIGURE 5.1

A simple REST interaction between user and server in HTTP specification.

(*Courtesy of Thomas Fielding [2]*)

is a resource. Any information that can be named can be a resource, such as a document or image or a temporal service. A resource is a conceptual mapping to a set of entities. Each particular resource is identified by a unique name, or more precisely, a *Uniform Resource Identifier (URI)* which is of type URL, providing a global addressing space for resources involved in an interaction between components as well as facilitating service discovery. The URIs can be bookmarked or exchanged via hyperlinks, providing more readability and the potential for advertisement.

Uniform, Constrained Interface: Interaction with RESTful web services is done via the HTTP standard, client/server cacheable protocol. Resources are manipulated using a fixed set of four CRUD (create, read, update, delete) verbs or operations: PUT, GET, POST, and DELETE. PUT creates a new resource, which can then be destroyed by using DELETE. GET retrieves the current state of a resource. POST transfers a new state onto a resource.

Self-Descriptive Message: A REST message includes enough information to describe how to process the message. This enables intermediaries to do more with the message without parsing the message contents. In REST, resources are decoupled from their representation so that their content can be accessed in a variety of standard formats (e.g., HTML, XML, MIME, plain text, PDF, JPEG, JSON, etc.). REST provides multiple/alternate representations of each resource. Metadata about the resource is available and can be used for various purposes, such as cache control, transmission error detection, authentication or authorization, and access control.

Stateless Interactions: The REST interactions are “stateless” in the sense that the meaning of a message does not depend on the state of the conversation. Stateless communications improve visibility, since a monitoring system does not have to look beyond a single request data field in order to determine the full nature of the request reliability as it facilitates the task of recovering from partial failures, and increases scalability as discarding state between requests allows the server component to quickly free resources. However, stateless interactions may decrease network performance by increasing the repetitive data (per-interaction overhead). Stateful interactions are based on the concept of explicit state transfer. Several techniques exist to exchange state, such as URI rewriting, cookies, and hidden form fields. State can be embedded in response messages to point to valid future states of the interaction.

Such lightweight infrastructure, where services can be built with minimal development tools, is inexpensive and easy to adopt. The effort required to build a client to interact with a RESTful service is rather small as developers can begin testing such services from an ordinary web browser, without having to develop custom client-side software. From an operational point of view, a stateless RESTful web service is scalable to serve a very large number of clients, as a result of REST support for caching, clustering, and load balancing.

RESTful web services can be considered an alternative to SOAP stack or “big web services,” described in the next section, because of their simplicity, lightweight nature, and integration with HTTP. With the help of URIs and hyperlinks, REST has shown that it is possible to discover web resources without an approach based on registration to a centralized repository. Recently, the web Application Description Language (WADL) [3] has been proposed as an XML vocabulary to describe RESTful web services, enabling them to be discovered and accessed immediately by potential clients. However, there are not a variety of toolkits for developing RESTful applications. Also, restrictions on GET length, which does not allow encoding of more than 4 KB of data in the

Table 5.1 REST Architectural Elements (Adapted from [2])

REST Elements	Elements	Example
Data elements	Resource	The intended conceptual target of a hypertext reference
	Resource identifier	URL
	Representation	HTML document, JPEG image, XML, etc.
	Representation metadata	Media type, last-modified time
	Resource metadata	Source link, alternates, vary
	Control data	If-modified-since, cache-control
Connectors	Client	libwww, libwww-perl
	Server	libwww, Apache API, NSAPI
	Cache	Browser cache, Akamai cache network
	Resolver	Bind (DNS lookup library)
	Tunnel	SSL after HTTP CONNECT
Components	Origin server	Apache httpd, Microsoft IIS
	Gateway	Squid, CGI, Reverse Proxy
	Proxy	CERN Proxy, Netscape Proxy, Gauntlet
	User agent	Netscape Navigator, Lynx, MOMspider

resource URI, can create problems because the server would reject such malformed URIs, or may even be subject to crashes. REST is not a standard. It is a design and architectural style for large-scale distributed systems.

Table 5.1 lists the REST architectural elements. Several Java frameworks have emerged to help with building RESTful web services. Restlet [4], a lightweight framework, implements REST architectural elements such as resources, representation, connector, and media type for any kind of RESTful system, including web services. In the Restlet framework, both the client and the server are components. Components communicate with each other via connectors.

JSR-311 (JAX-RS) [5], a specification provided by Sun Microsystems, defines a set of Java APIs for the development of RESTful web services. The specification provides a set of annotations with associated classes and interfaces that can be used to expose Java objects as web resources. JSR-311 provides clear mappings between the URI and corresponding resources, and mappings between HTTP methods with the methods in Java objects, by using annotations. The API supports a wide range of HTTP entity content types including HTML, XML, JSON, GIF, JPG, and so on. Jersey [6] is a reference implementation of the JSR-311 specification for building RESTful web services. It also provides an API for developers to extend Jersey based on their needs.

Example 5.1 RESTful Web Service in Amazon S3 Interface

A good example of RESTful web service application in high-performance computing systems is the Amazon Simple Storage Service (S3) interface. Amazon S3 is data storage for Internet applications. It provides simple web services to store and retrieve data from anywhere at any time via the web. S3 keeps fundamental entities, "objects," which are named pieces of data accompanied by some metadata to be

Table 5.2 Sample REST Request-Response for Creating an S3 Bucket

REST Request	REST Response
PUT/[bucket-name] HTTP/1.0 Date: Wed, 15 Mar 2011 14:45:15 GMT Authorization:AWS [aws-access-key-id]: [header-signature] Host: s3.amazonaws.com	HTTP/1.1 200 OK x-amz-id-2: VjzdTviQorQtSjcgLshzCZSzN+7CnewvHA +6sNxR3VRcUPyO5fmSmo8bWnIS52qa x-amz-request-id: 91A8CC60F9FC49E7 Date: Wed, 15 Mar 2010 14:45:20 GMT Location: /[bucket-name] Content-Length: 0 Connection: keep-alive

stored in containers called "buckets," each identified by a unique key. Buckets serve several purposes: They organize the Amazon S3 namespace at the highest level, identify the account responsible for storage and data transfer charges, play a role in access control, and serve as the unit of aggregation for usage reporting. Amazon S3 provides three types of resources: a list of user buckets, a particular bucket, and a particular S3 object, accessible through https://s3.amazonaws.com/{name-of-bucket}/{name-of-object}.

These resources are retrieved, created, or manipulated by basic HTTP standard operations: GET, HEAD, PUT, and DELETE. GET can be used to list buckets created by the user, objects kept inside a bucket, or an object's value and its related metadata. PUT can be used for creating a bucket or setting an object's value or metadata, DELETE for removing a particular bucket or object, and HEAD for getting a specific object's metadata. The Amazon S3 API supports the ability to find buckets, objects, and their related metadata; create new buckets; upload objects; and delete existing buckets and objects for the aforementioned operations. Table 5.2 shows some sample REST request-response message syntax for creating an S3 bucket.

Amazon S3 REST operations are HTTP requests to create, fetch, and delete buckets and objects. A typical REST operation consists of sending a single HTTP request to Amazon S3, followed by waiting for an HTTP response. Like any HTTP request, a request to Amazon S3 contains a request method, a URI, request headers which contain basic information about the request, and sometimes a query string and request body. The response contains a status code, response headers, and sometimes a response body.

The request consists of a PUT command followed by the bucket name created on S3. The Amazon S3 REST API uses the standard HTTP header to pass authentication information. The authorization header consists of an AWS Access Key ID and AWS SecretAccess Key, issued by the developers when they register to S3 Web Services, followed by a signature. To authenticate, the *AWSAccessKeyId* element identifies the secret key to compute the signature upon request from the developer. If the request signature matches the signature included, the requester is authorized and subsequently, the request is processed.

■

The composition of RESTful web services has been the main focus of composite Web 2.0 applications, such as mashups, to be discussed in Sections 5.4 and 9.1.3. A mashup application combines capabilities from existing web-based applications. A good example of a mashup is taking images from an online repository such as Flickr and overlaying them on Google Maps. Mashups differ from all-in-one software products in that instead of developing a new feature into an existing tool, they combine the existing tool with another tool that already has the desired feature. All tools work independently, but create a uniquely customized experience when used together in harmony.

5.1.2 Services and Web Services

In an SOA paradigm, software capabilities are delivered and consumed via loosely coupled, reusable, coarse-grained, discoverable, and self-contained services interacting via a message-based communication model. The web has becomes a medium for connecting remote clients with applications for years, and more recently, integrating applications across the Internet has gained in popularity. The term "web service" is often referred to a self-contained, self-describing, modular application designed to be used and accessible by other software applications across the web. Once a web service is deployed, other applications and other web services can discover and invoke the deployed service (Figure 5.2).

In fact, a web service is one of the most common instances of an SOA implementation. The W3C working group [1] defines a web service as a software system designed to support interoperable machine-to-machine interaction over a network. According to this definition, a web service has an interface described in a machine-executable format (specifically Web Services Description Language or WSDL). Other systems interact with the web service in a manner prescribed by its description using SOAP messages, typically conveyed using HTTP with an XML serialization in

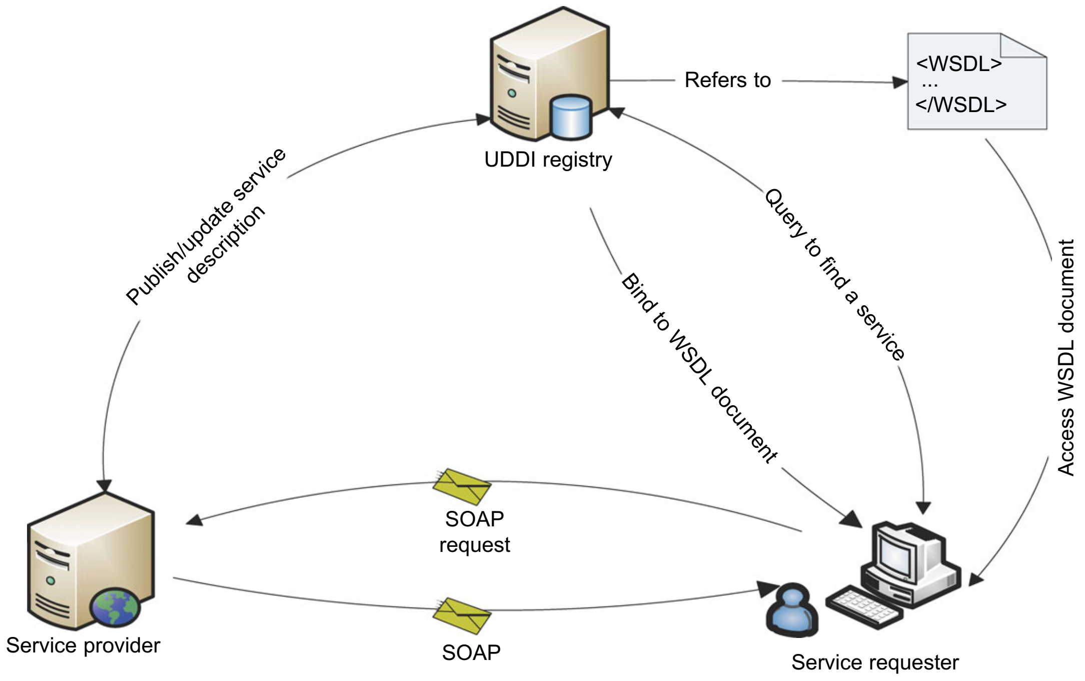

FIGURE 5.2

A simple web service interaction among provider, user, and the UDDI registry.

conjunction with other web-related standards. The technologies that make up the core of today's web services are as follows:

Simple Object Access Protocol (SOAP) SOAP provides a standard packaging structure for transmission of XML documents over various Internet protocols, such as SMTP, HTTP, and FTP. By having such a standard message format, heterogeneous middleware systems can achieve interoperability. A SOAP message consists of a root element called *envelope*, which contains a *header*: a container that can be extended by intermediaries with additional application-level elements such as routing information, authentication, transaction management, message parsing instructions, and Quality of Service (QoS) configurations, as well as a *body* element that carries the payload of the message. The content of the payload will be marshaled by the sender's SOAP engine and unmarshaled at the receiver side, based on the XML schema that describes the structure of the SOAP message.

Web Services Description Language (WSDL) WSDL describes the interface, a set of operations supported by a web service in a standard format. It standardizes the representation of input and output parameters of its operations as well as the service's protocol binding, the way in which the messages will be transferred on the wire. Using WSDL enables disparate clients to automatically understand how to interact with a web service.

Universal Description, Discovery, and Integration (UDDI) UDDI provides a global registry for advertising and discovery of web services, by searching for names, identifiers, categories, or the specification implemented by the web service. UDDI is explained in detail in Section 5.4.

SOAP is an extension, and an evolved version of XML-RPC, a simple and effective remote procedure call protocol which uses XML for encoding its calls and HTTP as a transport mechanism, introduced in 1999 [7]. According to its conventions, a procedure executed on the server and the value it returns was a formatted in XML. However, XML-RPC was not fully aligned with the latest XML standardization. Moreover, it did not allow developers to extend the request or response format of an XML-RPC call. As the XML schema became a W3C recommendation in 2001, SOAP mainly describes the protocols between interacting parties and leaves the data format of exchanging messages to XML schema to handle.

The major difference between web service technology and other technologies such as J2EE, CORBA, and CGI scripting is its standardization, since it is based on standardized XML, providing a language-neutral representation of data. Most web services transmit messages over HTTP, making them available as Internet-scale applications. In addition, unlike CORBA and J2EE, using HTTP as the tunneling protocol by web services enables remote communication through firewalls and proxies.

SOAP-based web services are also referred to as "big web services" [7]. As we saw earlier in this chapter, RESTful [8] services can also be considered a web service, in an HTTP context. SOAP-based web services interaction can be either synchronous or asynchronous, making them suitable for both request-response and one-way exchange patterns, thus increasing web service availability in case of failure.

5.1.2.1 WS-I Protocol Stack

Unlike RESTful web services that do not cover QoS and contractual properties, several optional specifications have been proposed for SOAP-based web services to define nonfunctional requirements and to guarantee a certain level of quality in message communication as well as reliable,

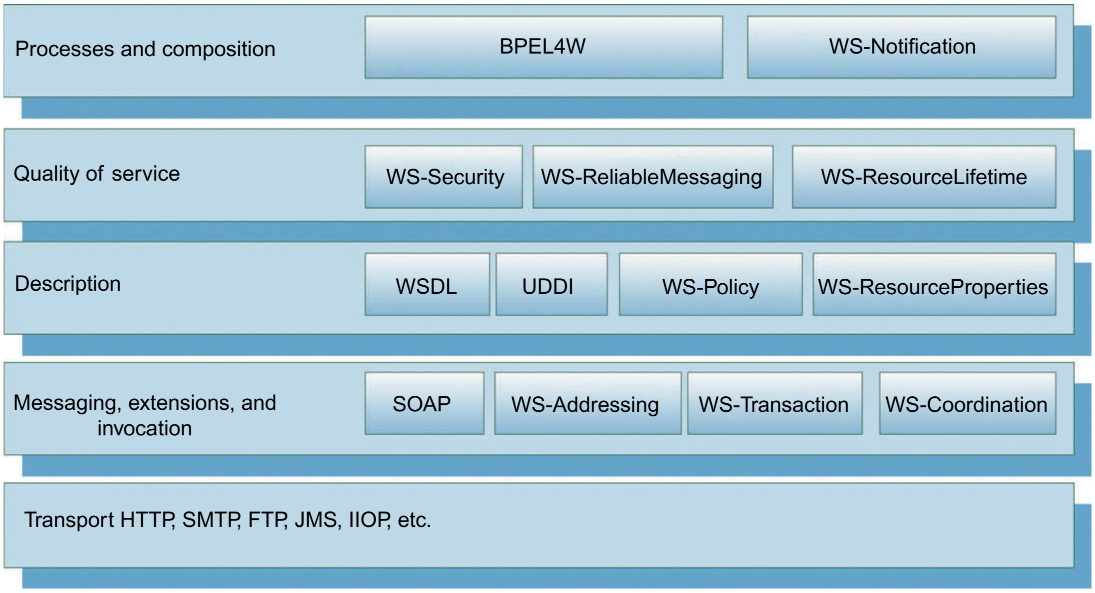

FIGURE 5.3

WS-I protocol stack and its related specifications.

transactional policies, such as WS-Security [9], WS-Agreement [10], WS-ReliableMessaging [11], WS-Transaction [12], and WS-Coordination [13] as shown in Figure 5.3.

As mentioned, SOAP messages are encoded using XML, which requires that all self-described data be sent as ASCII strings. The description takes the form of start and end tags which often constitute half or more of the message's bytes. Transmitting data using XML leads to a considerable transmission overhead, increasing the amount of transferred data by a factor 4 to 10 [14]. Moreover, XML processing is compute and memory intensive and grows with both the total size of the data and the number of data fields, making web services inappropriate for use by limited-profile devices, such as handheld PDAs and mobile phones.

Web services provide on-the-fly software composition, described further in Section 5.5, through the use of loosely coupled, reusable software components. By using Business Process Execution Language for Web Services (BPEL4WS), a standard executable language for specifying interactions between web services recommended by OASIS, web services can be composed together to make more complex web services and workflows. BPEL4WS is an XML-based language, built on top of web service specifications, which is used to define and manage long-lived service orchestrations or processes.

In BPEL, a business process is a large-grained stateful service, which executes steps to complete a business goal. That goal can be the completion of a business transaction, or fulfillment of the job of a service. The steps in the BPEL process execute activities (represented by BPEL language elements) to accomplish work. Those activities are centered on invoking partner services to perform tasks (their job) and return results back to the process. BPEL enables organizations to automate their business processes by orchestrating services. Workflow in a grid context is defined [15] as

"The automation of the processes, which involves the orchestration of a set of Grid services, agents and actors that must be combined together to solve a problem or to define a new service."

The JBPM [16] Project, built for the JBoss [17] open source middleware platform, is an example of a workflow management and business process execution system. Another workflow system, Taverna [18], has been extensively used in life science applications. There are a variety of tools for developing and deploying web services in different languages, among them SOAP engines such as Apache Axis for Java, gSOAP [19] for C++, the *Zolera Soap Infrastructure* (ZSI) [20] for Python, and Axis2/Java and Axis2/C. These toolkits, consisting of a SOAP engine and WSDL tools for generating client stubs, considerably hide the complexity of web service application development and integration. As there is no standard SOAP mapping for any of the aforementioned languages, two different implementations of SOAP may produce different encodings for the same objects.

Since SOAP can combine the strengths of XML and HTTP, as a standard transmission protocol for data, it is an attractive technology for heterogeneous distributed computing environments, such as grids and clouds, to ensure interoperability. As we discussed in Section 7.4, *Open Grid Services Architecture* (OGSA) grid services are extensions of web services and in new grid middleware, such as Globus Toolkit 4 and its latest released version GT5, pure standard web services. Amazon S3 as a cloud-based persistent storage service is accessible through both a SOAP and a REST interface. However, REST is the preferred mechanism for communicating with S3 due to the difficulties of processing large binary objects in the SOAP API, and in particular, the limitation that SOAP puts on the object size to be managed and processed. Table 5.3 depicts a sample SOAP request-response to get a user object from S3.

A SOAP message consists of an envelope used by the applications to enclose information that need to be sent. An envelope contains a header and a body block. The *EncodingStyle* element refers

Table 5.3 Sample SOAP Request-Response for Creating an S3 Bucket

SOAP Request	SOAP Response
<**soap:Envelope** xmlns:soap="http://www.w3.org/2003/05/soap-envelope" soap:encodingStyle= "http://www.w3.org/2001/12/soap-encoding"> <**soap:Body**> <**CreateBucket** xmlns="http://doc.s3.amazonaws .com/2010-03-15"> <**Bucket**>**SampleBucket**</**Bucket**> <AWSAccessKeyId> 1B9FVRAYCP1VJEXAMPLE= </AWSAccessKeyId> <Timestamp>2010-03-15T14:40:00.165Z </Timestamp> <Signature>Iuyz3d3P0aTou39dzbqaEXAMPLE =</Signature> </CreateBucket> </soap:Body> </soap:Envelope>	<**soap:Envelope** xmlns:soap="http://www.w3.org/2003/05/soap-envelope" soap:encodingStyle= "http://www.w3.org/2001/12/soap-encoding"> <**soap:Body**> <**CreateBucket** xmlns="http://doc.s3.amazonaws .com/2010-03-15"> <**Bucket**>**SampleBucket**</**Bucket**> <AWSAccessKeyId>1B9FVRAYCP1VJEXAMPLE= </AWSAccessKeyId> <Timestamp>2010-03-15T14:40:00.165Z </Timestamp> <Signature>Iuyz3d3P0aTou39dzbqaEXAMPLE =</Signature> </CreateBucket> </soap:Body> </soap:Envelope>

Table 5.4 The 10 Areas Covered by the Core WS-* Specifications

WS-* Specification Area	Examples
1. Core Service Model	XML, WSDL, SOAP
2. Service Internet	WS-Addressing, WS-MessageDelivery, Reliable WSRM, Efficient MOTM
3. Notification	WS-Notification, WS-Eventing (Publish-Subscribe)
4. Workflow and Transactions	BPEL, WS-Choreography, WS-Coordination
5. Security	WS-Security, WS-Trust, WS-Federation, SAML, WS-SecureConversation
6. Service Discovery	UDDI, WS-Discovery
7. System Metadata and State	WSRF, WS-MetadataExchange, WS-Context
8. Management	WSDM, WS-Management, WS-Transfer
9. Policy and Agreements	WS-Policy, WS-Agreement
10. Portals and User Interfaces	WSRP (Remote Portlets)

to the URI address of an XML schema for encoding elements of the message. Each element of a SOAP message may have a different encoding, but unless specified, the encoding of the whole message is as defined in the XML schema of the root element. The header is an optional part of a SOAP message that may contain auxiliary information as mentioned earlier, which does not exist in this example.

The body of a SOAP request-response message contains the main information of the conversation, formatted in one or more XML blocks. In this example, the client is calling *CreateBucket* of the Amazon S3 web service interface. In case of an error in service invocation, a SOAP message including a *Fault* element in the body will be forwarded to the service client as a response, as an indicator of a protocol-level error.

5.1.2.2 WS- Core SOAP Header Standards*

Table 5.4 summarizes some of the many (around 100) core SOAP header specifications. There are many categories and several overlapping standards in each category. Many are expanded in this chapter with XML, WSDL, SOAP, BPEL, WS-Security, UDDI, WSRF, and WSRP. The number and complexity of the WS-* standards have contributed to the growing trend of using REST and not web services. It was a brilliant idea to achieve interoperability through self-describing messages, but experience showed that it was too hard to build the required tooling with the required performance and short implementation time.

Example 5.2 WS-RM or WS-Reliable Messaging

WS-RM is one of the best developed of the so-called WS-* core web service specifications. WS-RM uses message instance counts to allow destination services to recognize message delivery faults (either missing or out-of-order messages). WS-RM somewhat duplicates the capabilities of TCP-IP in this regard, but operates at a different level—namely at the level of user messages and not TCP packets, and from source to destination independent of TCP routing in between. This idea was not fully developed (e.g., multicast messaging is not properly supported). Details can be found at http://en.wikipedia.org/wiki/WS-ReliableMessaging.

5.1.3 Enterprise Multitier Architecture

Enterprise applications often use multitier architecture to encapsulate and integrate various functionalities. Multitier architecture is a kind of client/server architecture in which the presentation, the application processing, and the data management are logically separate processes. The simplest known multilayer architecture is a two-tier or client/server system. This traditional two-tier, client/server model requires clustering and disaster recovery to ensure resiliency. While the use of fewer nodes in an enterprise simplifies manageability, change management is difficult as it requires servers to be taken offline for repair, upgrading, and new application deployments. Moreover, the deployment of new applications and enhancements is complex and time-consuming in fat-client environments, resulting in reduced availability. A three-tier information system consists of the following layers (Figure 5.4):

- **Presentation layer** Presents information to external entities and allows them to interact with the system by submitting operations and getting responses.

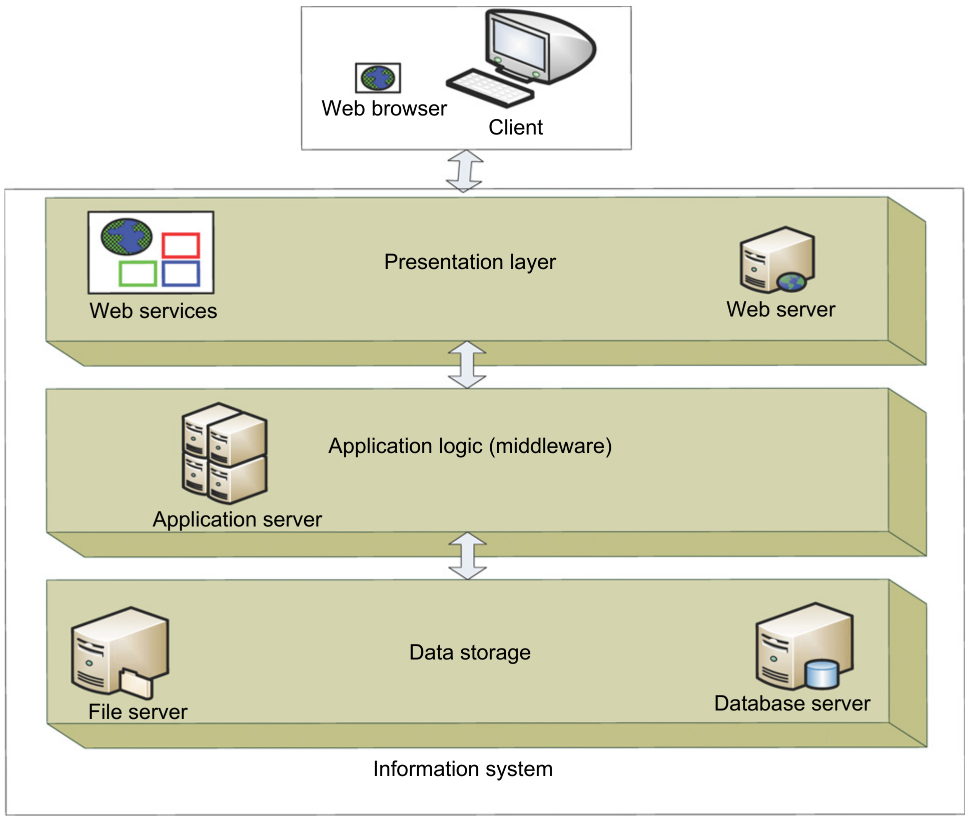

FIGURE 5.4

Three-tier system architecture.

(Gustavo Alonso, et al, Web Services: Concepts, Architectures and Applications (Data-Centric Systems and Applications). Springer Verlag, 2010. With kind permission from Springer Science+Business Media B.V.)

- **Business/application logic layer or middleware** Programs that implement the actual operations requested by the client through the presentation layer. The middle tier can also control user authentication and access to resources, as well as performing some of the query processing for the client, thus removing some of the load from the database servers.
- **Resource management layer** Also known as the data layer, deals with and implements the different data sources of an information system.

In fact, a three-tier system is an extension of two-tier architecture where the application logic is separated from the resource management layer [21]. By the late 1990s, as the Internet became an important part of many applications, the industry extended the three-tier model to an *N*-tier approach. SOAP-based and RESTful web services have become more integrated into applications. As a consequence, the data tier split into a data storage tier and a data access tier. In very sophisticated systems, an additional wrapper tier can be added to unify data access to both databases and web services. Web services can benefit from the separation of concerns inherent in multitier architecture in almost the same way as most dynamic web applications [22].

The business logic and data can be shared by both automated and GUI clients. The only differences are the nature of the client and the presentation layer of the middle tier. Moreover, separating business logic from data access enables database independence. *N*-tier architecture is characterized by the functional decomposition of applications, service components, and their distributed deployment. Such an architecture for both web services and dynamic web applications leads to reusability, simplicity, extensibility, and clear separation of component functionalities.

Web services can be seen as another tier on top of the middleware and application integration infrastructure [23], allowing systems to interact with a standard protocol across the Internet. Because each tier can be managed or scaled independently, flexibility is increased in the IT infrastructure that employs *N*-tier architecture. In the next section, we will describe OGSA, as multitier, service-oriented architecture for middleware which describes the capabilities of a grid computing environment and embodies web services to make computing resources accessible in large-scale heterogeneous environments.

5.1.4 Grid Services and OGSA

The OGSA [24], developed within the OGSA Working Group of the Global Grid Forum (recently renamed to Open Grid Forum or OGF and being merged with the Enterprise Grid Alliance or EGA in June 2006), is a service-oriented architecture that aims to define a common, standard, and open architecture for grid-based applications. "Open" refers to both the process to develop standards and the standards themselves. In OGSA, everything from registries, to computational tasks, to data resources is considered a service. These extensible set of services are the building blocks of an OGSA-based grid. OGSA is intended to:

- Facilitate use and management of resources across distributed, heterogeneous environments
- Deliver seamless QoS
- Define open, published interfaces in order to provide interoperability of diverse resources
- Exploit industry-standard integration technologies
- Develop standards that achieve interoperability

- Integrate, virtualize, and manage services and resources in a distributed, heterogeneous environment
- Deliver functionality as loosely coupled, interacting services aligned with industry-accepted web service standards

Based on OGSA, a grid is built from a small number of standards-based components, called grid services. [25] defines a grid service as "a web service that provides a set of well-defined interfaces, following specific conventions (expressed using WSDL)." OGSA gives a high-level architectural view of grid services and doesn't go into much detail when describing grid services. It basically outlines what a grid service should have. A grid service implements one or more interfaces, where each interface defines a set of operations that are invoked by exchanging a defined sequence of messages, based on the *Open Grid Services Infrastructure* (OGSI) [26]. OGSI, also developed by the Global Grid Forum, gives a formal and technical specification of a grid service.

Grid service interfaces correspond to *portType*s in WSDL. The set of *portType*s supported by a grid service, along with some additional information relating to versioning, are specified in the grid service's *serviceType*, a WSDL extensibility element defined by OGSA. The interfaces address discovery, dynamic service creation, lifetime management, notification, and manageability; whereas the conventions address naming and upgradeability. Grid service implementations can target native platform facilities for integration with, and of, existing IT infrastructures.

According to [25], OGSA services fall into seven broad areas, defined in terms of capabilities frequently required in a grid scenario. Figure 5.5 shows the OGSA architecture. These services are summarized as follows:

- **Infrastructure Services** Refer to a set of common functionalities, such as naming, typically required by higher level services.
- **Execution Management Services** Concerned with issues such as starting and managing tasks, including placement, provisioning, and life-cycle management. Tasks may range from simple jobs to complex workflows or composite services.
- **Data Management Services** Provide functionality to move data to where it is needed, maintain replicated copies, run queries and updates, and transform data into new formats. These services must handle issues such as data consistency, persistency, and integrity. An OGSA data service is a web service that implements one or more of the base data interfaces to enable access to, and management of, data resources in a distributed environment. The three base interfaces, *Data Access, Data Factory*, and *Data Management*, define basic operations for representing, accessing, creating, and managing data.
- **Resource Management Services** Provide management capabilities for grid resources: management of the resources themselves, management of the resources as grid components, and management of the OGSA infrastructure. For example, resources can be monitored, reserved, deployed, and configured as needed to meet application QoS requirements. It also requires an information model (semantics) and data model (representation) of the grid resources and services.
- **Security Services** Facilitate the enforcement of security-related policies within a (virtual) organization, and supports safe resource sharing. Authentication, authorization, and integrity assurance are essential functionalities provided by these services.
- **Information Services** Provide efficient production of, and access to, information about the grid and its constituent resources. The term "information" refers to dynamic data or events used for

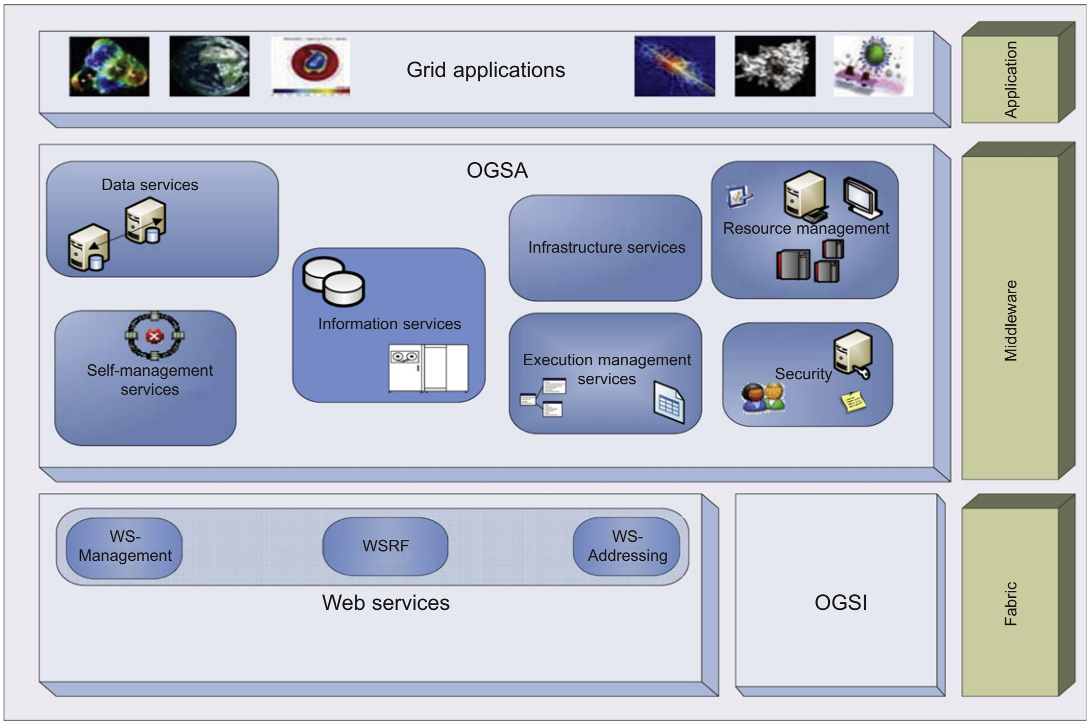

FIGURE 5.5

The OGSA architecture.

(*Courtesy of Foster, et al.* [24], *http://www.ogf.org/documents/GFD.80.pdf.*)

status monitoring; relatively static data used for discovery; and any data that is logged. Troubleshooting is just one of the possible uses for information provided by these services.

- **Self-Management Services** Support service-level attainment for a set of services (or resources), with as much automation as possible, to reduce the costs and complexity of managing the system. These services are essential in addressing the increasing complexity of owning and operating an IT infrastructure.

OGSA has been adopted as reference grid architecture by a number of grid projects. The first prototype grid service implementation was demonstrated January 29, 2002, at a Globus Toolkit tutorial held at Argonne National Laboratory. Since then, the Globus Toolkit 3.0 and 3.2 have offered an OGSA implementation based on OGSI. Two key properties of a grid service are transience and statefulness. Creation and destruction of a transient grid service can be done dynamically. The creation and lifetime of OGSA grid services are handled following the "factory pattern," to be explained in Section 7.3.1. Web service technologies are designed to support loosely coupled, coarse-grained dynamic systems, and hence do not meet all grid requirements, such as keeping state information, and thus they are unable to fully address the wide range of distributed systems OGSA is designed to support.

OGSA applies a set of WSDL extensions to represent the identifiers necessary to implement a grid service instance across any system. These extensions were defined by OGSI. A key extension is the grid service reference: a network-wide pointer to a specific grid service instance, which makes that instance accessible to remote client applications. These extensions, including the *Grid Service Handle* (GSH) and *Grid Service Reference* (GSR), will be described in Chapter 7. These extensions include stateful grid services and the shortcomings of OGSI with its dense and long specifications. Further problems concern incompatibility with some current web service tools and the fact that it takes a lot of concepts from object orientation.

Unlike the nature of web services, this has led to close cooperation between the grid and web service communities. As a result of these joint efforts, the *Web Services Resource Framework* (WSRF) [27], WS-Addressing [28], and *WS-Notification* (WSN) specifications have been proposed to OASIS. Consequently, OGSI extensions to web services have been deprecated in favor of new web service standards, and in particular, WSRF. WSRF is a collection of five different specifications. Of course, they all relate to the management of WS-Resources. Table 5.5 depicts WSRF-related interface operations.

Plain web services are usually stateless. This means the web service can't "remember" information, or keep state, from one invocation to another. However, since a web service is stateless, the following invocations have no idea of what was done in the previous invocations. Grid applications generally require web services to keep state information as they interact with the clients or other web services. The purpose of WSRF is to define a generic framework for modeling and accessing

Table 5.5 WSRF and Its Related Specifications

Specification			Description
WSRF Specifications	WS-ResourceProperties		Standardizes the definition of the resource properties, its association with the WS interface, and the messages defining the query and update capability against resource properties
	WS-ResourceLifetime		Provides standard mechanisms to manage the life cycle of WS-resources (e.g., setting termination time)
	WS-ServiceGroup		Standard expression of aggregating web services and WS-Resources
	WS-Basefault		Provides a standard way of reporting faults
WSRF-Related Specifications	WS-Notification	WS-Base Notification	Proposes a standard way of expressing the basic roles involved in web service publish and subscribe for notification message exchange
		WS-BrokeredNotification	Standardizes message exchanges involved in web service publish and subscribe of a message broker
		WS-Topics	Defines a mechanism to organize and categorize items of interest for a subscription known as "topics"
	WS-Addressing		Transport-neutral mechanisms to address web service and messages

persistent resources using web services in order to facilitate the definition and implementation of a service and the integration and management of multiple services. Note that “stateless” services can, in fact, remember state if that is carried in messages they receive. These could contain a token remembered in a cookie on the client side and a database or cache accessed by the service. Again, the user accessing a stateless service can establish state for the session through the user login that references permanent information stored in a database.

The state information of a web service is kept in a separate entity called a resource. A service may have more than one (singleton) resource, distinguished by assigning a unique key to each resource. Resources can be either in memory or persistent, stored in secondary storage such as a file or database. The pairing of a web service with a resource is called a WS-Resource. The preferred way of addressing a specific WS-Resource is to use the qualified *endpoint reference (EPR)* construct, proposed by the WS-Addressing specification. Resources store actual data items, referred to as resource properties. Resource properties are usually used to keep service data values, providing information on the current state of the service, or metadata about such values, or they may contain information required to manage the state, such as the time when the resource must be destroyed. Currently, the Globus Toolkit 4.0 provides a set of OGSA capabilities based on WSRF.

5.1.5 Other Service-Oriented Architectures and Systems

A survey of services and how they are used can be found in [29]. Here we give two examples: one of a system and one of a small grid.

Example 5.3 U.S. DoD Net-Centric Services

The U.S. military has introduced a set of so-called Net-Centric services that are to be used in Department of Defense (DoD) software systems to be used on the GiG – Global Information Grid. As shown in Table 5.6, these make different choices of services from OGSA. This is not a radically different architecture, but rather a different layering. Messaging is present in Table 5.6 but viewed as part of WS-* or a

Table 5.6 Core Global Information Grid Net-Centric Services

Service or Feature	Examples
Enterprise services management	Life-cycle management
Security; information assurance (IA)	Confidentiality, integrity, availability, reliability
Messaging	Publish-subscribe important
Discovery	Data and services
Mediation	Agents, brokering, transformation, aggregation
Collaboration	Synchronous and asynchronous
User assistance	Optimize Global Information Grid user experience
Storage	Retention, organization, and disposition of all forms of data
Application	Provisioning, operations, and maintenance
Environmental control services	Policy

higher application layer in OGSA. (see more details in http://www.ogf.org/gf/group_info/view.php?group=infod-wg for the INFOD standard) [29].

Example 5.4 Services in CICC—The Chemical Informatics Grid

The goal of this project is to support cluster analysis, data mining, and quantum simulation/first principles calculations on experimentally obtained data on small molecules with potential use in drug development. Small molecule data is gathered from NIH PubChem and DTP databases, with additional large molecule data available from service-wrapped databases such as the Varuna, Protein Data Bank, PDBBind, and MODB. NIH-funded High Throughput Screening centers are expected to deluge the PubChem database with assays of the next several years, making the automated organization and analysis of data essential.

Data analysis applications are interestingly combined with text analysis applications applied to journal and technical articles to make a comprehensive scientific environment. Workflow is a key part of this project as it encodes scientific use cases. Many CICC services are based on Cambridge University's WWMM project (www-pmr.ch.cam.ac.uk/wiki/Main_Page) led by Peter Rust. Table 5.7 shows the mix of system and application services used in CICC. This is a small project and larger grids would have many more services. Details can be found at the web site: http://www.chembiogrid.org.

Table 5.7 Services and Standards Used in CICC

Service Name	Description
Workflow/Monitoring/ Management Services	Uses Taverna from the UK e-Science Program/OMII or mashups written in scripting languages.
Authentication/ Authorization	Currently all services are openly available.
Registry and Discovery	Will inherit registry services from other grids.
Portal and Portlets	Use a JSR 168-based portal.
File Services	No specialized service. URLs are used for naming files and simple remote download. Services developed previously for other grids can be used for uploads.
NIH DTP Database Services	Access to the NIH Developmental Therapeutics Program (DTP)'s database of molecular screens against 60 cancer cell lines. This is a free service provided by the NIH and used by Chembiogrid.
PubMed Search Service	Searchable online database of medical journal articles. CICC develops harvesting services of the abstracts that can be combined with text analysis applications such as OSCAR3.
SPRESI Services	Clients/service proxies to the commercial SPRESI service (www.spresi.com/). This scientific database houses molecular and reaction data and references and patents.
Varuna Database Service	Molecular structure and more detailed information (such as force fields).
VOTables Data Tables Web Service	CICC-developed web service based on the National Virtual Observatory's VOTables XML format for tabular data.

Table 5.7 Services and Standards Used in CICC—cont'd

Service Name	Description
Specific Applications: BCI, OpenEye, Varuna, AutoGEFF	CICC inherits job management services from other grids (including one based on Apache Ant) for managing the execution of both commercial and in-house developed high-performance computing applications.
Condor and BirdBath	Examine the use of Condor and its SOAP interface (BirdBath) as a super-scheduler for Varuna applications on the TeraGrid.
ToxTree Service	Wraps an algorithm for estimating toxic hazards in a particular compound. Useful in combination with other clustering programs in a workflow.
OSCAR3 Service	Based on OSCAR3 by the WWMM group, performs text analysis on journal articles and other documents to extract (in XML) the chemistry-specific information. SMILES assigned to well-known compounds. Works with traditional database and clustering algorithms.
CDK Services	CICC has developed a number of simple services based on the Chemistry Development Kit (CDK). These include similarity calculations, molecular descriptor calculations, fingerprint generators, 2D image generators, and 3D coordinate molecular generators.
OpenBabel Service	Converts between various chemical formats (such as between InChI and SMILES).
InChIGoogle	For a given InChI (a string specification of a molecular structure), performs a Google search to return a page-ranked list of matches.
Key Interfaces/Standards/Software Used	WSDL, SOAP (with Axis 1.x). CML, InChI, SMILES, Taverna SCUFl, JSR-168 JDBC Servlets, VOTables
Unused Interfaces/Software	WS-Security, JSDL, WSRF, BPEL, OGSA-DAI

5.2 MESSAGE-ORIENTED MIDDLEWARE

This section introduces message-oriented middleware for supporting distributed computing. The study included enterprise bus, publish-subscribe model, queuing, and messaging systems.

5.2.1 Enterprise Bus

In the previous section, we described services and service architectures. These services, by definition, interact with messages with a variety of different formats (APIs), wire protocols, and transport mechanisms. It is attractive to abstract the communication mechanism so that services can be defined that communicate independent of details of the implementation. For example, the author of a service should not need to worry that a special port is to be used to avoid firewall difficulties or that we need to use UDP and special fault tolerance approaches to achieve satisfactory latency on a long-distance communication. Further, one may wish to introduce a wrapper so that services expecting

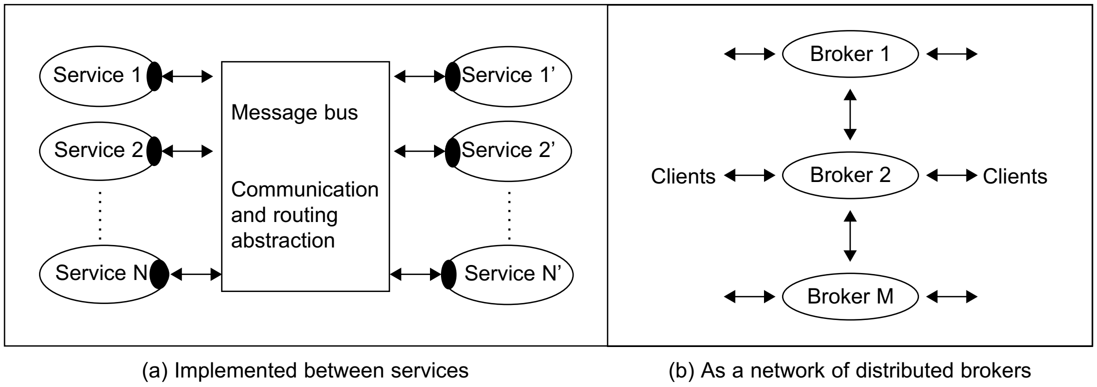
Service 1
Service 2
Service N
Message bus
Communication and routing abstraction
Service 1'
Service 2'
Service N'
(a) Implemented between services
Broker 1
Broker 2
Broker M
Clients
Clients
(b) As a network of distributed brokers

FIGURE 5.6

Two message bus implementations between services or using a broker network.

messages in different styles (say, SOAP, REST, or Java RMI) can communicate with each other. The term "enterprise service bus" or ESB [30,31] refers to the case where the bus supports the convenient integration of many components, often in different styles. These remarks motivate the messaging black box abstraction shown in Figure 5.6.

One does not open a channel between source and destination, but rather injects a message into the bus with enough information to allow it to be delivered correctly. This injection is performed by code loaded into each service and represented by the filled ovals as client interfaces in Figure 5.6(a). The message bus is shown linking services in this figure, but it can work with any software or hardware entity sending and receiving messages. A simple example could be desktops or smart phones as the clients. Further, such buses can be implemented internally to an application, or in a distributed fashion. In the latter case, the message bus is typically implemented as a set of "brokers" shown in Figure 5.6(b).

The use of multiple brokers allows the bus to scale to many clients (services) and large message traffic. Note that the brokers of Figure 5.6(b) are "just" special servers/services that receive messages and perform needed transformations and routing on them and send out new messages. There is a special (simple) case of message buses where the brokers shown in Figure 5.6(b) are not separate servers but are included in the client software. Note that such buses support not just point-to-point messaging but broadcast or selective multicast to many recipient clients (services).

Often, one implements brokers as managers of queues, and software in this area often has MQ or "Message Queue" in its description. An early important example is MQSeries [32] from IBM which is now marketed as the more recent WebSphereMQ [32,33]. Later, when we study cloud platforms in Chapter 8, we will find that both Azure and Amazon offer basic queuing software. A typical use of a message queue is to relate the master and workers in the "farm" model of parallel computing where the "master" defines separate work items that are placed in a queue which is accessed by multiple workers that select the next available item. This provides a simple dynamically load-balanced parallel execution model. If necessary, the multiple brokers of Figure 5.6(b) can be used to achieve scalability.

5.2.2 Publish-Subscribe Model and Notification

An important concept here is "publish-subscribe" [34] which describes a particular model for linking source and destination for a message bus. Here the producer of the message (publisher) labels the message in some fashion; often this is done by associating one or more topic names from a (controlled) vocabulary. Then the receivers of the message (subscriber) will specify the topics for which they wish to receive associated messages. Alternatively, one can use content-based delivery systems where the content is queried in some format such as SQL.

The use of topic or content-based message selection is termed message filtering. Note that in each of these cases, we find a many-to-many relationship between publishers and subscribers. Publish-subscribe messaging middleware allows straightforward implementation of notification or event-based programming models. The messages could, for example, be labeled by the desired notifying topic (e.g., an error or completion code) and contain content elaborating the notification [34].

5.2.3 Queuing and Messaging Systems

There are several useful standards in this field. The best known is the Java Message Service (JMS) [35] which specifies a set of interfaces outlining the communication semantics in pub/sub and queuing systems. *Advanced Message Queuing Protocol (AMQP)* [36] specifies the set of wire formats for communications; unlike APIs, wire formats are cross-platform. In the web service arena, there are competing standards, WS-Eventing and WS-Notification, but neither has developed a strong following. Table 5.8 compares a few common messaging and queuing systems. We selected two cloud systems: Amazon Simple Queue and Azure Queue.

We also list MuleMQ [37], which is the messaging framework underlying the ESB [30,31] system Mule, developed in Java, of which there are 2,500 product deployments as of 2010. The focus of Mule is to simplify the integration of existing systems developed using JMS, Web Services, SOAP, JDBC, and traditional HTTP. Protocols supported within Mule include POP, IMAP, FTP, RMI, SOAP, SSL, and SMTP. ActiveMQ [38] is a popular Apache open source message broker while WebSphereMQ [33] is IBM's enterprise message bus offering. Finally, we list the open source NaradaBrokering [39] that is notable for its broad range of supported transports and was successfully used to support a software *Multipoint Control Unit (MCU)* for multipoint video conferencing and other collaboration capabilities.

Note that the four noncloud systems support JMS. Also, some key features of messaging systems are listed in the table but are not discussed in this brief section. These are security approach and guarantees and mechanisms for message delivery. Time-decoupled delivery refers to situations where the producer and consumer do not have to be present at the same time to exchange messages. Fault tolerance is also an important property: Some messaging systems can back up messages and provide definitive guarantees. This table is only illustrative and there are many other important messaging systems. For example, RabbitMQ [40] is a new impressive system based on the AMQP standard.

5.2.4 Cloud or Grid Middleware Applications

Three examples are given here to illustrate the use of the NaradaBrokering middleware service with distributed computing. The first example is related to environmental protection. The second is for Internet conferencing and the third is for earthquake science applications.

Table 5.8 Comparison of Selected Messaging and Queuing Systems

System Features	Amazon Simple Queue [41]	Azure Queue [42]	ActiveMQ	MuleMQ	WebSphere MQ	Narada Brokering
AMQP compliant	No	No	No, uses OpenWire and Stomp	No	No	No
JMS compliant	No	No	Yes	Yes	Yes	Yes
Distributed broker	No	No	Yes	Yes	Yes	Yes
Delivery guarantees	Message retained in queue for four days	Message accessible for seven days	Based on journaling and JDBC drivers to databases	Disk store uses one file/channel, TTL purges messages	Exactly-once delivery supported	Guaranteed and exactly-once
Ordering guarantees	Best effort, once delivery, duplicate messages exist	No ordering, message returns more than once	Publisher order guarantee	Not clear	Publisher order guarantee	Publisher- or time-order by Network Time Protocol
Access model	SOAP, HTTP-based GET/POST	HTTP REST interfaces	Using JMS classes	JMS, Adm. API, and JNDI	Message Queue Interface, JMS	JMS, WS-Eventing
Max. message	8 KB	8 KB	N/A	N/A	N/A	N/A
Buffering	N/A	Yes	Yes	Yes	Yes	Yes
Time decoupled delivery	Up to four days; supports timeouts	Up to seven days	Yes	Yes	Yes	Yes
Security scheme	Based on HMAC-SHA1 signature, Support for WS-Security 1.0	Access to queues by HMAC SHA256 signature	Authorization based on JAAS for authentication	Access control, authentication, SSL for communication	SSL, end-to-end application-level data security	SSL, end-to-end application-level data security, and ACLs
Support for web services	SOAP-based interactions	REST interfaces	REST	REST	REST, SOAP interactions	WS-Eventing
Transports	HTTP/HTTPS, SSL	HTTP/HTTPS	TCP, UDP, SSL, HTTP/S, Multicast, in-VM, JXTA	Mule ESB supports TCP, UDP, RMI, SSL, SMTP, and FTP	TCP, UDP, Multicast, SSL, HTTP/S	TCP, Parallel TCP, UDP, Multicast, SSL, HTTP/S, IPsec
Subscription formats	Access is to individual queues	Access is to individual queues	JMS spec allows for SQL selectors; also access to individual queues	JMS spec allows for SQL selectors; also access to individual queues	JMS spec allows SQL selectors; access to individual queues	SQL selectors, regular expressions, <tag, value> pairs, XQuery and XPath

Example 5.5 Environmental Monitoring and Internet Conference Using NaradaBrokering

The GOAT project at Clemson University is part of the Program of Integrated Study for Coastal Environmental Sustainability (PISCES), which addresses environmental sustainability issues that can accompany coastal development. The current study incorporates groundwater monitoring, surface water quality and quantity monitoring, weather, and a variety of ecological measurements. The project utilizes the publish-subscribe messaging system, NaradaBrokering, to provide a flexible and reliable layer to move observation data from a wide variety of sensor sources to users that have diverse data management and processing requirements. NaradaBrokering can display environmental sensors.

The commercial Internet Meeting software Anabas (www.anabas.com) incorporates support for sharing applications besides incorporating support for shared whiteboards, and chat tools. Anabas uses NaradaBrokering for its content dissemination and messaging requirements. On a daily basis, Anabas supports several online meetings in the United States, Hong Kong, and mainland China. Note NaradaBrokering supports audio-video conferencing (using UDP) as well as other collaborative applications using TCP. Dynamic screen display published to NaradaBrokering can be displayed on collaborating clients.

Example 5.6 QuakeSim Project for Earthquake Science

The NASA-funded QuakeSim project uses NaradaBrokering to manage workflows that connect distributed services, and to support GPS filters delivering real-time GPS data to both human and application consumers as shown in Figure 5.7 (http://quakesim.jpl.nasa.gov/). The GPS application is an important

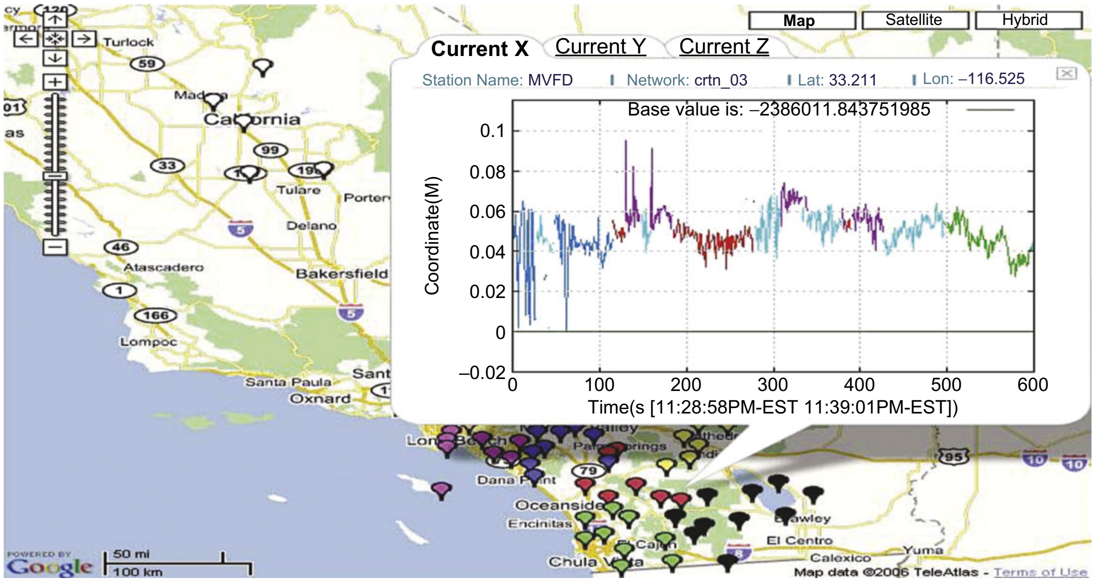

FIGURE 5.7

Display of GPS sensors managed by NaradaBrokering in Southern California; the map displays the time series produced by one of the GPS stations. (http://quakesim.jpl.nasa.gov/).

application similar to Example 5.4 where publish-subscribe systems manage sensor networks. In fact, one can consider webcams as sensors (as they produce real-time streams) and so Example 5.5 is also of this type. Clouds are an important implementation for this type of application as brokers can be added on demand to support myriad dynamic sensors from cell phones to military or civil sensors responding to an extreme event. The display of GPS sensors is managed by NaradaBrokering. The map displays the time series produced by GPS stations.

5.3 PORTALS AND SCIENCE GATEWAYS

Science gateways [43,44] are tools that enable interactive, web-based science, education, and collaboration. Gateways provide user-centric environments for interacting with remote computational resources through user interfaces that are typically (but not exclusively) built with web technologies. Although they have superficial similarities to web sites and may leverage tools such as content management systems for their presentation layers, gateways are much more complicated entities. Science gateways are also termed portals. This section describes the general architecture of gateways, surveys several prominent examples, and discusses example gateway-building software.

We can categorize gateway software as "turnkey" solutions, exemplified by HUBzero; and "toolbox" solutions, exemplified by the Open Gateway Computing Environments project. Turnkey gateway software provides an end-to-end solution for building gateways, including hosting. Toolbox gateway software provides tools that solve specific problems and can be integrated into customized software stacks. The SimpleGrid project [45] is another toolkit example. HUBzero and OGCE software will be described in more detail shortly.

Figure 5.8 provides a high-level overview of the grouping of components that make up a gateway. The bottom tier is the resource layer, which may include campus computing clusters and

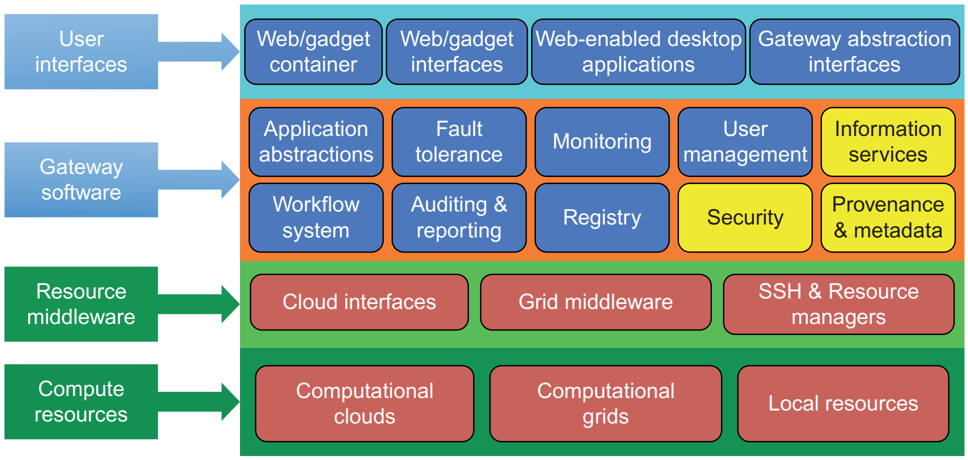

FIGURE 5.8

A gateway component software stack for scientific applications.

storage systems, national-scale grid resources such as the TeraGrid [46] and the Open Science Grid [47], and computational cloud resources. The second tier consists of middleware for interacting with these resources. Common middleware examples include Globus [48] (providing job execution and remote file management), Condor [49] (providing job scheduling), and iRods [50] (providing data and metadata movement and management). Middleware systems typically expose secure, network-accessible APIs and developer libraries as well as providing numerous command-line tools.

Tier 2 software and services run on the resources of Tier 1. The next two tiers are not tied to these resources and can run on separate host machines. Tier 3 is the gateway software layer and consists of service components that manage the execution of scientific applications and data on the Tier 1 resources through the Tier 2 interfaces. Execution management for both data and science applications can be decoupled into several components: the initial invocation, monitoring, fault tolerance, and task coupling as workflows. Security considerations [51,52] (such as authentication and authorization) permeate all tiers, but we locate the component in Tier 3 since the user's identity is established here.

We also place user and group management (including social networking) in this tier, along with third-party information services [53]. Finally, the top tier (Tier 4) is the user presentation layer. Presentation layers may be built with a number of different tools, including content management systems, Open Social gadgets, and desktop applications, as described in more detail shortly. Figure 5.8 is flexible enough to describe most gateways. We next make the description more concrete by reviewing two example science gateways: GridChem and UltraScan. Both gateways have been used for significant scientific research.

5.3.1 Science Gateway Exemplars

Three examples are given here to illustrate the gateway concept in scientific applications using computational grids or Internet clouds.

Example 5.7 Computational Chemistry Grid (CCG)

The CCG is also known as GridChem (www.gridchem.org) [54,55], which is one of the most heavily used Science Gateways in TeraGrid. GridChem provides a grid-enabled desktop interface that enables users to set up, launch, and manage computational chemistry simulations on the TeraGrid. Applications supported by GridChem include Gaussian, CHARMM, and GAMESS.

These are computationally demanding parallel applications that require supercomputers, although supporting applications—input validation and visualization tools—can run on the user's desktop. GridChem pioneered gateway concepts such as "community users," which allow users to access resources through a shared allocation, that are now commonplace in many Science Gateways. GridChem has delivered significant scientific accomplishments in the form of scientific publications that acknowledge and highlight the GridChem cyber-infrastructure's utility for computational chemistry [56].

Example 5.8 UltraScan Biophysics Gateway

UltraScan [57,58] has developed a TeraGrid Science Gateway for the high-resolution analysis and modeling of hydrodynamic data from an analytical ultracentrifuge (AUC). This application is used for the solution-based study of biological macromolecules and synthetic polymers by biochemists, biophysicists, and material scientists for fitting experimental data to finite element solutions of flow equations. More than 700 biochemists, biophysicists, biologists, and material scientists worldwide rely on UltraScan software for analysis of their experimental data. The software has assisted in understanding an array of disease processes, including cancer, neurodegenerative diseases, HIV/AIDS, diabetes, Huntington's disease, and aging studies. By conservative estimates, UltraScan has contributed to more than 250 peer-reviewed publications including 23 known publications from 2009 [59].

UltraScan makes its core experimental analysis software available as an online service to scientific users through its Science Gateway. This analysis software is computationally demanding and must be run on clusters and supercomputers. UltraScan job management services (Tier 3) hide complexities and provide fault tolerance for the experimental scientists using the portal. Although it uses TeraGrid for some of its computing power, UltraScan needs to span multiple resource providers: It also uses university clusters and would like to extend its resources to include international grids in Germany and Australia. The key to UltraScan's success and growth over the next three years is to provide the ability to manage jobs across multiple cyberinfrastructure resources that are not collectively part of a single, managed grid.

Gateways such as UltraScan provide an example where code optimization and efficient use of compute resources are done by experts in the field and shared with hundreds of end users. The gateways lower the entry barrier for analyzing the data on high-end resources. As an example, the UltraScan gateway provides an optimal solution for solving large non-negative least squares problems that arise in data analysis [60,61]. Solving these problems requires significant compute resources. The procedure improves compute resource utilization targeting the inverse problem involved in modeling analytical ultracentrifugation experimental data. Solving large non-negatively constrained least squares systems is frequently used in the physical sciences to estimate model parameters which best fit experimental data.

AUC is an important hydrodynamic experimental technique used in biophysics to characterize macromolecules and to determine parameters such as molecular weight and shape. Newer AUC instruments equipped with *multi-wavelength* (MWL) detectors have recently increased the data sizes by three orders of magnitude. Analyzing the MWL data requires significant compute resources. UltraScan gateway bridges these requirements for end users by providing both the procedures and the capabilities to execute them on supercomputing resources.

Example 5.9 The nanoHUB.org Gateway

The nanoHUB.org gateway is operated by the National Science Foundation (NSF)-funded Network for Computational Nanotechnology (NCN) to support the National Nanotechnology Initiative in the United States and worldwide and to accelerate the transformation of nanoscience into nanotechnology. Since 2002, the community using nanoHUB.org has grown from 1,000 users mostly at Purdue University to more than 290,000 visitors each year from 172 countries worldwide. From August 2009 to July 2010, some 8,600 users accessed more than 170 nanotechnology simulation tools and launched 340,000 simulation runs.

Today nanoHUB.org hosts more than 2,000 content items such as 170 online simulation tools and 43 complete courses, plus tutorials, research seminars, and teaching materials. All of nanoHUB.org's services are free of charge to the user. All resources on nanoHUB.org are presented in scholarly form with title, authors, abstract, and archival citation information. To date, there are more than 560 citations in academic literature to nanoHUB.org and the tools, seminars, and other resources published there. Also to date, 379 courses at 131 institutions of higher education have used nanoHUB.org resources. The journal citations and the documented use in classrooms is evidence that nanoHUB.org resources aid both research and education.

5.3.2 HUBzero Platform for Scientific Collaboration

HUBzero is an open source software platform used to create web sites or "hubs" for scientific collaboration, research, and education.[1] It has a unique combination of capabilities that appeals to many people engaged in research and educational activities. Like YouTube.com, HUBzero allows people to upload content and "publish" to a wide audience, but instead of being restricted to short video clips, it handles many different kinds of scientific content. In that respect, HUBzero resembles MIT's OpenCourseWare [64], but it also integrates the content with collaboration capabilities. Like Google Groups, HUBzero lets people work together in a private space where they can share documents and send messages to one another. Like Askville on Amazon.com, HUBzero lets people ask questions and post responses, but about scientific concepts instead of products.

Perhaps the most interesting feature of HUBzero is the way it handles simulation and modeling programs, or "tools." Like SourceForge.net, HUBzero allows researchers to work collaboratively on the source code of their simulation programs and share those programs with the community. But instead of sharing only by offering source code bundles to download, HUBzero also offers live *published* programs available for use instantly and entirely within an ordinary web browser. The simulation engines run start to finish on computational resources selected for that hub.

Computationally demanding runs can be dispatched to remote resources in a way that is completely transparent to users. Tools are driven by friendly GUIs that enable end-to-end operation of the simulation process encompassing setup, execution, and data visualization. Many GUIs are built using HUBzero's Rappture toolkit, which lets researchers compare simulation results from multiple runs and ask "What if?" questions. In effect, each hub powered by HUBzero is an "app store" for a scientific community connected to a cloud of resources for app execution, complete with a library of training materials and other collaboration features to support app use.

HUBzero was created by Purdue University and the NSF-funded NCN to power its web site at nanoHUB.org [62,63]. Today, the same HUBzero software powers 30 similar gateways covering a wide variety of disciplines in engineering and science. Here are three examples:

- **GlobalHUB.org** *29,000 active users, online since December 2007* Leverages the group functionality within HUBzero to support engineering education on a global scale. Teams of students work together in groups on a variety of engineering projects.
- **cceHUB.org** *2,400 active users, online since June 2008* Approaches cancer care from an "engineering" perspective by gathering a database of blood samples from patients, extracting proteomic/metabolomic data, mining the data to find biomarker patterns, and modeling the efficacy of cancer treatments.

[1]HUBzero is a trademark of Purdue University.

- **NEES.org** *15,000 users, online since August 2010* Home of the NSF Network for Earthquake Engineering Simulation, which catalogs experimental data from 14 institutions that are simulating earthquake conditions in the lab. The site also hosts modeling tools used to visualize and analyze the data [64a].

5.3.2.1 The HUBzero Architecture

On the surface, each HUBzero-powered gateway is a web site built with the widely used, open source "LAMP" architecture—a Linux operating system, an Apache web server, a MySQL database, and PHP web scripting. HUBzero adds a scientific content management system, the open source Rappture toolkit used to create GUIs for simulation programs, and unique middleware for hosting simulation tools and scientific data, as shown in Figure 5.9.

Each tool description page includes a "launch" button. When a user presses that button, the middleware allocates a session container on the tool execution host, starts the X11 windowing system within the container, starts the tool, and connects the session back to the user's web browser via *Virtual Network Computing* (VNC) [65]. To the user, it appears that the tool is running in the browser, but it actually runs within the hub environment, where it has access to local computation and visualization clusters and remote computing resources such as TeraGrid, Open Science Grid, and Purdue's DiaGrid [66]. Sessions can be shared among users collaborating offline or in real time, and they are persistent, so the user can close the browser window and revisit the same session later.

Unlike other portals and cyber environments, the tools in a hub are interactive and engaging. Users can zoom in on a graph, rotate a molecule, and probe the isosurfaces in a 3D volume [67]—all interactively, without having to wait for a web page to refresh. Users can visualize results without reserving time on a supercomputer or waiting for a batch job to engage. Each hub can host an unlimited number of tools uploaded by its community members, and the tools are deployed without having to rewrite code for the web. The computational demand of these tools

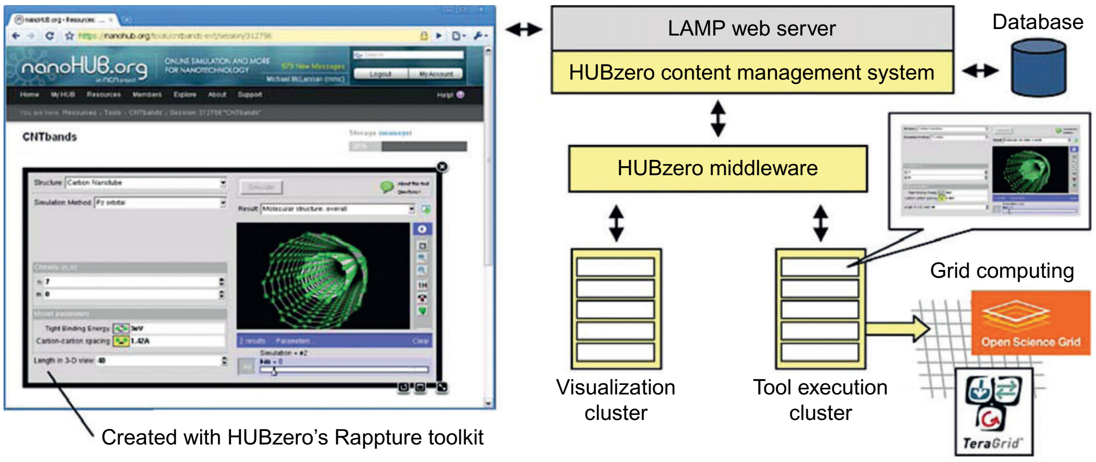

FIGURE 5.9

The HUBzero architecture and its major functional components.

may vary from seconds on a single core to hours and days on multiple, possibly large numbers of cores [68,69].

Each session runs in a restricted virtual environment implemented using OpenVZ [70], which controls access to file systems, networking, and other system resources. Users can see their files and processes, but cannot see others on the system, cannot launch attacks on other hosts, and cannot exceed their file quota. If a tool already has a GUI that runs under Linux/X11, it can be deployed as is in a matter of hours. If not, tool developers can use HUBzero's Rappture toolkit (http://rappture.org) to create a GUI with very little effort. Developers start by defining the inputs and outputs for their tool as a hierarchy of data objects written in an XML format. The data objects include simple elements, such as integers, Boolean values, numbers with units, and sets of mutually exclusive choices, as well as more complex objects, such as physical structures, finite element meshes, and molecules.

Rappture reads the XML description of a tool and generates a standard GUI for the tool automatically. The GUI prompts for values, launches simulation jobs, and loads results for visualization. The underlying simulation code uses an API to retrieve input values and store output results. Rappture includes libraries for C/C++, FORTRAN, MATLAB, Java, Python, Perl, Ruby, and Tcl/Tk, so the underlying simulator is not constrained to a particular language, but can be written in the developer's language of choice. The tool shown in Figure 5.9, for example, is a Rappture GUI for a MATLAB program that simulates carbon nanotubes.

5.3.2.2 Operational Features

Having instant access to a large collection of simulation tools opens up new capabilities, but also introduces a new set of issues. Users want to know more about the tools and the underlying physics they encode. They may discover a bug and wish to report it or question whether results are correct. They want to exchange ideas about new models and new features within the platform. HUBzero has evolved beyond a simple repository to include many features supporting these sorts of collaborations.

5.3.2.3 Ratings and Citations

The hub philosophy is not to judge the quality of each resource before it is posted, but rather to post resources and help the community assess the quality. Registered users are allowed to post five-star ratings and reviews for each resource. Users can also post citations of each resource in academic literature. The ratings and citations for each resource are combined with web analytics (measuring popularity) to produce a single number on a scale of 0 to 10, called the ranking, to represent the quality of the resource.

5.3.2.4 Content Tagging

Items on a hub are categorized by a series of tags, in much the same way that photos are tagged on a photo sharing site. This makes it easy to browse categories of resources, or to find resources that intersect two or more categories.

5.3.2.5 User Support Area

From time to time, users will have problems with logins, have questions about tools, and may otherwise need assistance. The HUBzero software comes with a built-in user support area. Users can click on the Help or Support link near the top of any page and fill out a form to file a support ticket. Tickets can be handled by the staff managing the hub, or forwarded to the researchers supporting various simulation tools. Some questions are beyond the understanding of the hub staff, or even beyond the knowledge of a single researcher.

HUBzero includes a question-and-answer forum patterned after Askville on Amazon.com that can engage the whole community. Any registered user can post a question, and other users can provide answers. At some point, the best answer is chosen as the "final" answer by the person who asked the question, and participants earn points as a reward for their effort. Points can be used as bragging rights, or as currency to purchase t-shirts and other merchandise in a hub store. The list of past questions/answers forms a knowledge base upon which a community can draw for immediate help with a similar problem.

Other issues are beyond software problems and physics questions, but really entail requests for tool improvements and new features. Such requests are tracked on a "wish list" for each tool and also on a wish list for the entire hub. All tickets, questions, and wishes are managed by the HUBzero software. Various administrators, software developers, and community members are given access to these facilities to help address the community support issues.

5.3.2.6 Wikis and Blogs

Each hub supports the creation of "topic" pages, which are wiki pages with a specific list of authors. Other users can add comments to a topic page and may suggest changes, which the original authors can choose to approve. Users can be added as coauthors for a page, so they can make further changes without approval. Ownership of a page can also be given to the entire community, so anyone can make changes without approval, in a wiki-like manner.

5.3.2.7 Usage Metrics

Each hub reports extensive metrics about how its resources are being used, just a few of which include the total number of users in a given period, the number of web hits, the number of simulation jobs launched, and CPU hours used. Metrics are reported down to the level of each individual resource, so everyone can see how many users have accessed a particular tool, or how many times a seminar has been viewed. Usage numbers are rolled up to provide an overview of usage for interesting categories, such as the total number of users that have accessed all resources published by a particular author. These usage metrics provide an incentive to use HUBzero-powered science gateways.

5.3.2.8 Future Directions

The development of HUBzero is driven largely by the projects using it. Although HUBzero started with a strong emphasis on simulation and modeling, it is evolving to include data management capabilities as well. Projects such as the cancer care engineering cceHUB.org and the earthquake engineering NEES.org are creating mechanisms for users to define, upload, publish, annotate, and analyze various types of structured data sets. Projects such as GlobalHUB.org are improving the group spaces where people exchange files and work together in a private context.

As hubs gather more tools, researchers see the need for stringing tools together in a workflow to tackle larger questions. A pill dissolution profile generated by one tool on pharmaHUB.org, for example, could be fed into a model of a patient's digestive track, and those results could be used to compute the amount of active ingredient in the patient's bloodstream over time. That chain of tools may need to be run hundreds of times to perform an overall sensitivity analysis, to optimize a design goal, or to quantify uncertainty in the output. A hub should not only catalog tools, but connect them one to another, to grid computing resources, and ultimately, to the researcher with a question to answer.

5.3.3 Open Gateway Computing Environments (OGCE)

The OGCE project [71] is a provider of open source gateway software that is used in several collaborating gateways [72]. OGCE consists of components that can be either used individually or integrated together to provide more comprehensive solutions for remote scientific application management. OGCE component tools include the following:

- **OGCE Gadget container [73]**, a Google tool for integrating user interface components
- **XRegistry**, a registry service for storing information about other online services and workflows
- **XBaya [74]**, a workflow composer and enactment engine
- **GFAC [75]**, a factory service that can be used to wrap command-line-driven science applications and make them into robust, network-accessible services
- **OGCE Messaging Service** supports events and notifications across multiple cooperating services

OGCE's strategy is based on the toolkit model. This strategy has been shaped by the TeraGrid science gateway program and its wide variety of gateways. There are obviously many frameworks, programming languages, and tools for building web-based gateways and providing advanced capabilities. It has been our experience that many gateways benefit from lightly coupled tools that can be taken collectively or in part. In the toolkit model, gateways can take a specific tool or tools and integrate them into the gateway's existing infrastructure. For example, a gateway such as UltraScan benefits from a reliable job submission tool that hides differences between Globus GRAM versions; GFAC is a candidate tool. GridChem and ParamChem want to extend their job submission capabilities to include scientific workflows; XBaya and its supporting tools can be used.

The OGCE tools focus on scientific application and workflow management and defer issues such as data and metadata management to other projects. There is significant variation in the characteristics exhibited by the scientific applications, so intermediate, application-specific services between the user interfaces and the generic grid middleware are desirable. The gateway software layer (Tier 3 of Figure 5.8) must accommodate the complexities of a particular domain and provide a software infrastructure that bridges the gap between the user interface layer and the grid middleware.

Consequently, many science gateways use scientific web services along with workflow systems to build the gateway software. Application-specific web services provide the bridge between generic grid middleware and the specific needs of a gateway. Workflows go a step further by assembling multiple services and steps into scientific use cases. Workflows may be implicit in the design of the gateway or they may be explicitly exposed to users. Registries are services that can be used to look up other services and workflows. Finally, messaging systems are used for different distributed components to communicate. For example, users need a mechanism for monitoring a long-running workflow.

5.3.3.1 Workflows

The OGCE scientific workflow system [74,76,77] provides a programming model that allows the scientist to program experiments using application web services developed with the GFAC service that abstract the complexities of the underlying middleware. The workflow system enables scientists to author experiments that can be stored, replayed, and shared with others. The workflow suite is bundled with interfaces for composition, execution, and monitoring of the workflows. Salient features include support for long-running applications and steering/dynamic user interactions.

OGCE's software stack is designed to be flexible in its coupling of various components to harness end-to-end, multiscale gateway infrastructures. Individual OGCE tools can be integrated into gateway deployments; likewise, other standard specification-based tools can be swapped within OGCE's software stack. As a specific example, the OGCE workflow system provides (with its XBaya frontend) a graphical interface to browse various application service registries (such as the OGCE's XRegistry).

From these registries, users can construct task graphs as workflows. The representation is captured as an abstract, high-level, workflow-neutral format, which can be translated into workflow execution specific syntax. Currently, integrations of BPEL [78], Jython, Taverna SCUFL [79], and Pegasus DAX [80] have all been demonstrated and exist at various levels of support. By default, the workflow enactment is facilitated by an open source BPEL implementation, the Apache *Orchestration and Director Engine* (ODE) [81], that OGCE developers have enhanced to support long-running scientific workflows on computational grids.

5.3.3.2 Scientific Application Management

Wrapping scientific applications as remotely accessible services is a common task for science gateways. Gateways that wish to offer many application services need a simple way to quickly wrap these applications. Task wrapping is only one step, however, in making an application available through a gateway. Grid middleware solutions (Tier 2 of Figure 5.8) abstract heterogeneous resources and offer a single unified job management interface to queuing systems. However, both the resources and grid middleware are still highly complex, so it is a difficult task to provide production-quality scientific application services that have the reliability and scalability (in number of users) needed by successful gateways. Gateways have often reinvented solutions for all of these problems.

The OGCE's GFAC tool is designed to wrap command-line-driven executables and make them available as external services. The primary goal of GFAC is to provide a general solution to the application-wrapping problem that can be reused by gateways as a plug-in service in their deployed infrastructure. GFAC generates web services that can be accessed through clients written in Java, Perl, PHP, Python, and other languages. GFAC-wrapped services can be run as stand-alone tools, or they can be registered with XRegistry for later incorporation into workflows. GFAC supports both persistent and dynamically created services.

Gateways adopting GFAC can outsource reliability and scalability issues and devote more resources to their domain-specific problems. The OGCE gateway suite has improved fault tolerance of computational jobs and data movement thanks to extensive efforts by the *Linked Environments for Atmospheric Discovery* (LEAD) Science Gateway [82]. The OGCE team has also leveraged large-scale coordinated gateway debugging efforts and applied the improvements to the advanced support requested gateways. Generic reliability and fault tolerance will not cover all problems: Codes will fail to run for reasons outside the gateway developer's control, so error detection, logging, and resolution strategies must be specialized to the application. Providing extensibility for gateways to address their application-specific error conditions is an upcoming effort for GFAC development.

5.3.3.3 Gadget Container

Workflows, registries, and service wrappers have both client- and server-side pieces. The OGCE builds most of its default user interfaces as gadgets. Gadgets are client-side web components (Tier 4 of Figure 5.8) that rely on HTML, CSS, and JavaScript rather than on specific server-side development frameworks. Gadgets for science gateways need to communicate with server-side components (Tier 3 of Figure 5.8). This can be done using REST services. Greater interactivity, simpler development, and the freedom to use a wide range of server-side tools make gadgets an interesting component model for science gateways.

The OGCE Gadget Container is an Open Social-compatible tool for aggregating web gadgets. Gadgets are mostly self-contained web applications that comply with Google gadget or Open Social standards. The gadget container provides an operating context for a particular gadget, which may reside on a completely separate web server. The OGCE Gadget Container provides layout and skin management and user-level customization. The container supports an OpenID authentication option for registered users and OAuth authorization for gadgets. The container runs under HTTPS security and also supports secure (HTTPS) connections between the container and gadgets.

The gadget container is built on top of Apache Shindig, the reference implementation for the Open Social standard. This allows the container to create Open Social-compatible social networks. The container also supports Google FriendConnect, which provides a simplified programming interface for social networks.

5.3.3.4 Packaging

OGCE software is open source and available for download through SourceForge, and plans are underway to start an Apache Incubator project for GFAC, XBaya, and supporting components. The preferred download mechanism is by SVN client checkouts and updates of tagged releases. The current OGCE release bridges several component projects. Each subproject can be built separately using Apache Maven; a master Maven POM is used to build all subprojects. This approach simplifies both development and deployment. Subprojects can be added when they mature, replaced with major upgrades, and discarded. Updates can be applied to specific components without rebuilding the entire software stack or developing a specific patching system. The OGCE software stack is designed to be portable and to compile on many platforms.

Building a science gateway requires to match end-user requirements, that is, the scientific use cases to be supported, to the capabilities of the gateway stack. Furthermore, developing the underlying workflows of the gateway is itself a lengthy scientific task, apart from the implementation issues. There are always components that can be used as is (security credential management and file browsers, for example), but OGCE software is intended to be extended and modified by gateway developers.

This requires a close collaboration between science domain experts and cyberinfrastructure specialists. Long-term sustainability is an important challenge facing all gateways; particularly those that depend on external resource providers such as the TeraGrid and Open Science Grid (see Figure 5.8 Tiers 1 and 2). These resources and their middleware evolve; gateways with Tier 3 and 4 components that are not actively maintained will decay. The challenge for many gateways is to maintain their middleware with reduced funding as the gateway matures from active development to stable usage.

5.4 DISCOVERY, REGISTRIES, METADATA, AND DATABASES

Distributed applications need to discover resources that suit their needs and manage them. In SOA, business services need to discover appropriate services to use and to integrate with. Registries are sophisticated naming and directory services that facilitate service resource discovery both at design and dynamically at runtime by classifying and categorizing services or metadata information about services. A registry requires a set of data structure specifications for the metadata to be stored in the registry, and a set of operations such as *Create, Read, Update, and Delete* (CRUD) for storing, deleting, and querying the data to store metadata for ownership, containment, and categorization of services. Registries usually contain three categories of information:

- **White pages** contain name and general contact information about an entity.
- **Yellow pages** contain classification information about the types and location of the services the entry offers.
- **Green pages** contain information about the details of how to invoke the offered services (technical data regarding the service).

Apart from registries, metadata, or information about data, can be used to facilitate discovery of resources and desired services. Metadata can be stored in relational or XML databases as metadata catalogs or added to web services to enhance service discovery capabilities. Integrating publish/subscribe patterns into databases can even add discovery capabilities to the static nature of databases, thus reducing the load on a single database caused by polling of enormous applications.

5.4.1 UDDI and Service Registries

UDDI specifications [83] define a way to describe, publish, and discover information about web services by creating a platform-independent, open framework. UDDI provides a name service and a directory service for looking up service descriptions by name or by a specific attribute. It was initiated in September 2000 as a joint collaboration on B2B integration between IBM, Microsoft, and Ariba. UDDI version 3.0 has been published as OASIS specifications and it has become an OASIS standard for public service registries.

The UDDI specification is focused on the definition of a collection of services supporting the description and discovery of: Businesses, organizations, and other web services providers; the web services they make available; and the technical interfaces which may be used to access those services. Based on a common set of industry standards, including HTTP, XML, XML Schema, and SOAP, UDDI provides an interoperable, foundational infrastructure for a web service-based software environment for both publicly available services and services only exposed internally within an organization.

There are two primary types of registries. A public registry is a logically centralized distributed service that replicates data with other public registries on a regular basis. A private registry is only accessible within a single organization or is shared by a group of business partners for a special purpose. The latter is also called a semiprivate or shared registry. The UDDI Business Registry consists of replicated registries (initially hosted by IBM and Microsoft) called UDDI operators.

A UDDI registry is an instance of a web service, and its entries can be published and queried using a SOAP-based interface. UDDI defines data structures and APIs for programmatically publishing service descriptions and querying the registry. Data in a UDDI registry is organized as instance types:

- ***businessEntity*** Describes an organization or a business that provides the web services, including the company name, contact information, industry/product/geographic classification, and so on
- ***businessService*** Describes a collection of related instances of web services offered by an organization, such as the name of the service, a description, and so forth
- ***bindingTemplate*** Describes the technical information necessary to use a particular web service, such as the URL address to access the web service instance and references to its description
- ***tModel*** A generic container for specification of WSDL documents in general web services
- ***publisherAssertion*** Defines a relationship between two or more *businessEntity* elements
- ***subscription*** A standing request to keep track of changes to the entities in the subscription

The entities *businessEntity, businessService, bindingTemplate*, and *tModel* form the core data structures of UDDI, each of which can be uniquely identified and accessed by a URI, called the "UDDI key." These entities and their relationships are depicted in Figure 5.10. A UDDI registry can be used by service providers, service requestors, or other registries. For such interactions with

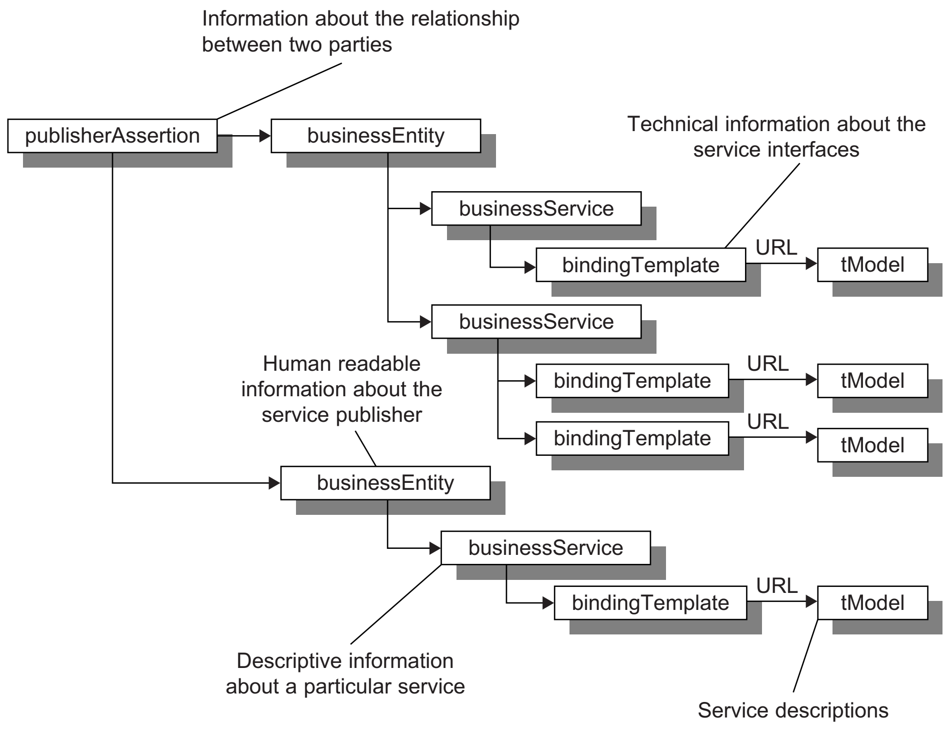
Information about the relationship between two parties
publisherAssertion
businessEntity
Technical information about the service interfaces
businessService
bindingTemplate
URL
tModel
businessService
Human readable information about the service publisher
bindingTemplate
URL
tModel
bindingTemplate
URL
tModel
businessEntity
businessService
bindingTemplate
URL
tModel
Descriptive information about a particular service
Service descriptions

FIGURE 5.10

UDDI entities and their relationship.

the registry, UDDI provides a set of APIs. Two basic types of API operations are applied to UDDI components:

- **UDDI Inquiry API** In order to find the set of registry entries such as business, service, binding, or *tMode*, details matching a particular search criterion (*find_*) or details of an entry corresponding to a given UDDI key (*get_*) operation can be used.
- **UDDI Publishers API** This enables add, modify, and delete entries by providing *save_* and *delete_* operations. In addition to the aforementioned look-up APIs, UDDI also defines general-purpose operation types, such as the next 4 specialized APIs.
- **UDDI Security API** Allows users to get and discard authentication tokens (*get_autToken, discard_autToken*)
- **UDDI Custody and Ownership Transfer API** Enables registries to transfer the custody of information among themselves and to transfer ownership of these structures one another (*transfer_entities*, *transfer_custody*)
- **UDDI Subscription API** Enables monitoring of changes in a registry by subscribing to track new, modified, and deleted entries (*delete_subscription, get_subscriptionResults, get_subscriptions, save_subscriptions*)
- **UDDI Replication API** Supports replication of information between registries so that different registries can be kept synchronized

Although UDDI is an open standard, it has never gained much popularity among various enterprise and scientific communities, as there has been no global registry for registering enterprise, e-science, or grid services following the UDDI specification after the closure of the public nodes of the Universal Business Registry operated by IBM, Microsoft, and SAP in January 2006. However, several public registries have been launched for public use by various communities providing a categorized list of a variety of services and related APIs. One of them is ProgrammableWeb.com [84].

ProgrammableWeb.com is a registry of a variety of Web 2.0 applications, such as mashups and APIs organized by category, date, or popularity. It has similar goals to UDDI, but does not use the detailed UDDI specifications. Mashups are composite Web 2.0 applications which combine capabilities from existing web-based applications, typically RESTful web services. Mashups can be compared to workflows, as they both implement distributed programming at the service level. Content used in mashups is typically sourced from a third party via a public interface or API. According to data released by the ProgrammableWeb.com register [84], most mashup and APIs are applied in mapping, search, travel, social, instant messaging, shopping, video streaming areas.

Other methods of sourcing content for mashups include web feeds (such as RSS) and JavaScript. web developers are able to programmatically search and retrieve APIs, mashups, member profiles, and other data from the ProgrammableWeb.com catalog, integrate on-demand registry and repository functionality into any service, and dynamically add new content as well as comment on existing entries, using the provided API, based on open standards including XML, RSS, OpenSearch, and *Atom Publishing Protocol* (APP) [85]. Among the most popular mashups frequently used on the web site are those provided for Google Maps, Flickr, Facebook, Twitter, and YouTube.

5.4.2 Databases and Publish-Subscribe

Publish-subscribe is a design pattern that enables asynchronous interaction among distributed applications and was discussed in Section 5.2 for middleware. Many high-level applications regularly query the database in order to adapt their execution according to this information. Such periodic data polling is not only inefficient and unscalable, but also resource-demanding on both sides, especially when the interval between calls to the database is very small, or in cases when there is more than one consumer application, it may increase the amount of network traffic and CPU usage dramatically. The publish-subscribe mechanism, already largely adopted in the implementation of today's applications, solves this issue. In a publish-subscribe interaction, event subscribers register to particular event types and receive notifications from the event publishers when they generate such events.

There is a dynamic, many-to-many relationship between event publishers and event subscribers, as there can be any number of publishers/subscribers for any type of event which can vary at any time. Publish-subscription adds dynamicity to static the nature of databases. While the publish-subscribe pattern was first implemented in centralized client/server systems, current research focuses mainly on distributed versions. The key benefit of the distributed publish-subscribe mechanism is the natural decoupling of publishers and subscribers. Since the publishers are unconcerned with the potential consumers of their data, and the subscribers are unconcerned with the locations of the potential producers of interesting data, the client interface of the publish-subscribe system is simple and intuitive.

Publish-subscribe systems are classified as either topic-based or content-based. In topic-based systems, publishers generate events with respect to a topic or subject. Subscribers then specify their interest in a particular topic, and receive all events published on that topic. Defining events in terms of topic names only is inflexible and requires subscribers to filter events belonging to general topics. Content-based systems solve this problem by introducing a subscription scheme based on the contents of events. Content-based systems are preferable as they give users the ability to express their interest by specifying predicates over the values of a number of well-defined attributes. The matching of publications (events) to subscriptions (interest) is done based on the content. Distributed solutions are mainly focused on topic-based publish-subscribe systems

Database systems provide many features that a messaging-based architecture can exploit, such as reliable storage, transactions, and triggers. On the other hand, integrated publish-subscribe capabilities in the database account for information-sharing systems that are simpler to deploy and maintain. However, since publish-subscribe and database technology have evolved independently, designing and implementing database-publish-subscribe-aware systems requires bringing together concepts and functionality from two separate worlds.

Jean Bacon, et al. [86] have extended the PostgreSQL open source database system to include publish-subscribe middleware functionality. It is based on the integration of active databases and the publish-subscribe communication model to form a global event-based system: Databases define and advertise change events, and clients subscribe to events of interest, and can refine their subscriptions through content-based filter expressions. This allows a database system in the local domain to function as an event broker (broker), reliably routing events among publishers, subscribers, and other brokers. This integration simplifies information management by grouping security, configuration (e.g., type schema), and recovery tasks for database and pub/sub operations under the same interface. Aktas has described use of publish-subscribe in grid information systems [87–89].

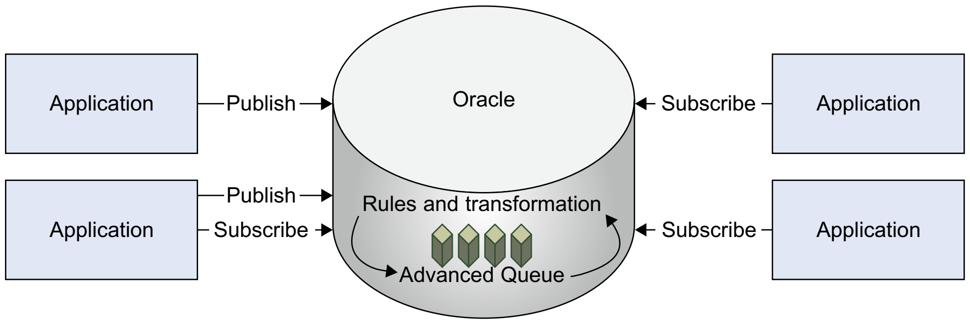

FIGURE 5.11

Oracle publish-subscribe model.

Message queuing and publish-subscribe are tightly intertwined: Oracle introduced a publish-subscribe solution for enterprise information delivery and messaging, shown in Figure 5.11, and based on Advanced Queuing—an implementation based on JMS and fully integrated in the Oracle database—to publish data changes and to automate business process workflows for distributed applications. Ontologies and other semantic mechanisms can be used to make events in the system more "context-aware." Advanced Queuing was introduced in Oracle 8 and extended in Oracle 9i to support publish-subscription. In Oracle release 10.1, Advanced Queuing was integrated into Oracle Streams, called Oracle Streams Advanced Queuing.

A combination of features are introduced to allow a publish-subscribe style of messaging between applications. These features include rule-based subscribers, message propagation, the listen feature, and notification capabilities. Oracle Streams Advanced Queuing is built on top of Oracle Streams and leverages the functionality of Oracle Database so that messages can be stored persistently, propagated between queues on different computers and databases, and transmitted using Oracle Net Services and HTTP(S). As Oracle Streams Advanced Queuing is implemented in database tables, all operational benefits of *high availability* (HA), scalability, and reliability are also applicable to queue data. Standard database features such as recovery, restart, and security are supported by Oracle Streams Advanced Queuing. Database development and management tools such as Oracle Enterprise Manager can be applied to monitor queues. Like other database tables, queue tables can be imported and exported.

5.4.3 Metadata Catalogs

Metadata catalogs play a vital role in distributed heterogeneous environments such as grids by providing users and applications the means to discover and locate the desired data and services among lots of sites on such environments. Metadata is information about data. Metadata is important, since it adds context to the data, in order to identify, locate, and interpret it. Key metadata on the grid includes the name and location of the data resource, structure of the data held within the data resource, data item names and descriptions, and user information (name, address, and profiles and preferences), or basic listings and simple lookup of available services, relating function and location without significant rich context. Metadata catalogs are used by various groups and communities, ranging from high-energy physics to biomedical, earth observation, and geological science.

Because of the importance of metadata services for the use of large-scale local or wide area storage resources, many groups have made efforts to investigate and implement such services. Among the earliest metadata catalogs is the *Metadata Catalog Service* (MCAT) [90], which is a part of the *Storage Resource Broker (SRB)* [91], which has evolved to the iRODS system [50]. MCAT, developed by the San Diego Supercomputing Center, aims to provide an abstraction layer over heterogeneous storage devices and file systems either inside or across computing centers. MCAT stores the data hierarchically using a tree of collections and is both a file and metadata catalog. Later versions of MCAT support replication and federation of data resources.

The MCS developed by the Globus Alliance [92] provides hierarchical organization of metadata and flexible schemas, and hides the storage backend from the user. The Globus project also contains the *Replica Location Service (RLS)* [93], which uses index servers to provide a global list of files available on different replica catalogs. Several LHC experiments have implemented their own specific metadata catalogs using a standard relational database backend and providing an intermediate layer to access the catalog on a distributed environment.

AMGA (the *ARDA Metadata for Grid Applications*) [94] is the official metadata catalog of the gLite software stack of the EGEE project. It began as an exploratory project to study the metadata requirements of the LHC experiments, and has since been deployed by several groups from different user communities, including high-energy physics (for LHCb bookkeeping), biomedicine, and earth observation. AMGA uses a hierarchical file system-like model to structure metadata, stored in a relational database. It stores entries representing the entities that are being described, such as files. The entries are grouped into collections, which can have a variant number of user-defined attributes, called the schema of the collection.

Attributes are represented as key-value pairs with type information, and each entry assigns an individual value to the attributes of its collection. A schema can be a representation of a directory, which can contain either entries or other schemas. As an advantage of this tree-like structure, users can define a hierarchical structure which can help to better organize metadata in subtrees that can be queried independently. The server supports several storage systems by using modules. AMGA can manage groups of users with different permissions on directories. In grid environment, file and metadata catalogs are used by users for discovering and locating data among the hundreds of grid sites.

5.4.4 Semantic Web and Grid

The grid strives to share and access metadata in order to automate information discovery and integration of services and resources in a dynamic, large-scale distributed environment. Meanwhile, the Semantic web is all about automation discovery and integration: adding machine-processable semantics to data so that computers can understand such information and process it on behalf of end users, thus enabling more intelligent web searching and linkage based on attaching rich metadata to web pages. The Semantic web aims to provide an environment where software agents are able to dynamically discover, interrogate, and interoperate resources and perform sophisticated tasks on behalf of humans, which is not far from the ambition of grid computing.

To achieve this, work has been undertaken to assert the meaning of web resources in a common data model, the *Resource Description Framework (RDF)*, using agreed ontologies expressed in a common language, such as the OWL web ontology language, so we can share the metadata and add in background knowledge. From this basis we should be able to query, filter, integrate, and aggregate the metadata, and reason over it to infer more metadata by applying rules and policies.

RDF is the first language developed for the Semantic web, using XML to represent information (including metadata) about resources on the web. RDF uses web identifiers (URIs), and describes resources in terms of simple properties and property values. OWL is an expressive ontology language that extends RDF schema. OWL adds more vocabulary for describing properties and classes: among others, relations between classes, cardinality, equality, richer typing of properties, characteristics of properties, and enumerated classes.

Semantic web services describe and annotate various aspects of a web service using explicit, machine-understandable semantics, facilitating the discovery, execution, monitoring, and aggregation of resources and web services, which solves interoperability issues and helps bring resources together to create virtual organizations. The OWL-S ontology enables web services to be described semantically and their descriptions to be processed and understood by software agents. It provides a standard vocabulary that can be used together with the other aspects of the OWL description language to create service descriptions. The OWL-S ontology defines the top-level concept of a "Service" and three OWL-S subontologies:

- **Service profile** Expresses what a service does in order to enable service advertisement and discovery.
- **Service model** Describes how the service works in order to enable service invocation, composition, monitoring, and recovery.
- **Service grounding** Specifies the details of how to access the service. A grounding can be thought of as a mapping from an abstract to a concrete specification, based on WSDL as a particular specification language.

The "semantic grid" or "grid with semantics" aims to incorporate the advantages of the grid, Semantic web, and web services. Built on the W3C Semantic Web Initiative, it is an extension of the current grid in which information and services are given well-defined meanings (ontologies, annotations, and negotiation processes as studied in the Semantic web and Software Agent paradigms), better enabling computers and people to work in cooperation. The semantic grid provides a general semantic-based, computational, and knowledge-based service for enabling the management and sharing of complex resources and reasoning mechanisms, thus systematically exposing semantically rich information associated with resources to build more intelligent grid services.

The notion of the semantic grid shown in Figures 5.12 and 5.13 was first articulated in the context of e-science, which is all about scientific investigation among scientists of various communities such as physicists, biologist, and chemists and their resources performed through distributed global collaborations such as the grid to solve scientific problems by generating, analyzing, sharing, and discussing their insights, experiments, and results in an effective manner, and the computing infrastructure that enables this joint effort. Higher-level services use the information relating to the resources' capability and the mechanisms for service fulfillment to automatically discover interoperable services and select the most appropriate service for the user with minimal human intervention.

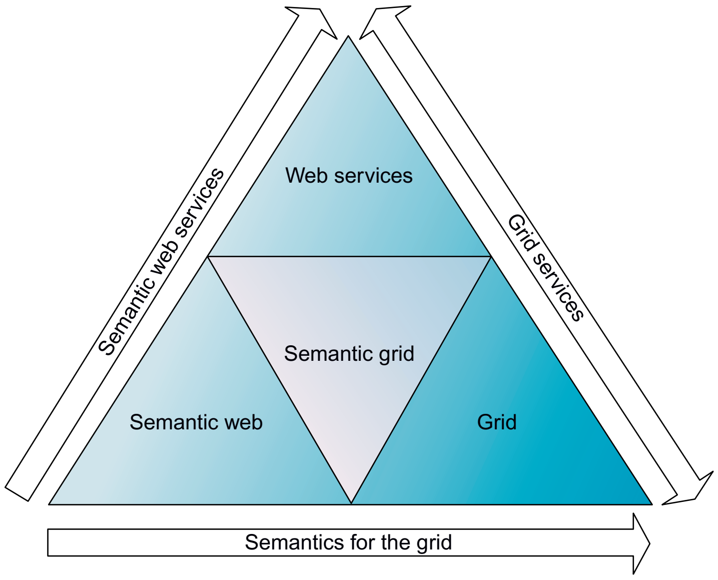

FIGURE 5.12

Semantic grid-related concepts and technologies.

(*Courtesy of Goble and Roure, (ECAI-2004), Valencia, Spain, [95]*)

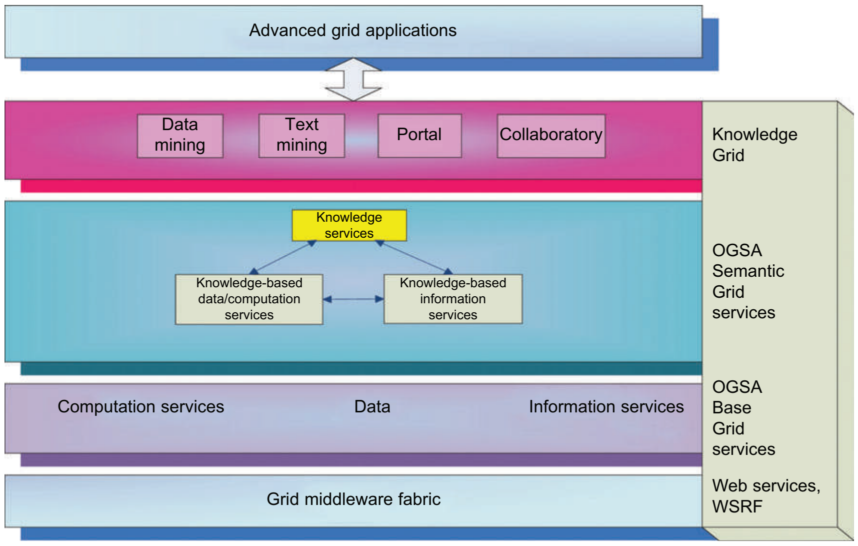

FIGURE 5.13

Semantic grid architecture.

(*Courtesy of Goble and Roure, (ECAI-2004), Valencia, Spain, [95]*) Semantic OGSA (S-OGSA) [96] has been proposed as reference architecture for semantic-based grids. S-OGSA extends OGSA to support the explicit handling of semantics, and defines the associated knowledge services to support a set of service capabilities. This is done by introducing semantic provisioning services that give support to the provision of semantics, by allowing the creation, storage, update, removal, and access of different forms of knowledge and metadata. This set of services includes ontology management and reasoning services, metadata services, and annotation services

S-OGSA has three main aspects: the model (the elements which it is composed of and their interrelationships), the capabilities (the services needed to deal with such components), and the mechanisms (the elements that will enable delivery when deploying the architecture in an application, grounded to a grid platform). The myGrid [97] Project community as a pioneer of semantic grid efforts focusing on the challenges of e-science has developed a suite of tools and services to enable workflow-based composition of diverse biological data and computational resources. Within the project, Semantic web technologies have been applied to the problems of resource discovery and workflow results management in a grid environment.

5.4.5 Job Execution Environments and Monitoring

A distributed job execution environment often consists of two components: a job execution engine and a distributed data management system. The job execution engine mainly deals with job scheduling, resource allocation, and other issues such as fault tolerance. The data management system often provides an abstraction for jobs to access distributed data. In recent years, to tackle the increasing need for Internet scale information management and processing, many Internet service companies built their own distributed systems for their specific needs. Most of these systems provide distributed execution engines and support integrated applications. Google MapReduce [98] and Microsoft Dryad [99] are two examples of this type of system described in Chapter 6.

MapReduce was primarily designed to support Google applications that use and generate large data sets. It generalizes a map/reduce abstraction from these applications and provides a simple programming model for distributed execution of programs. Communications between programs are generalized as exchanges of key-value pairs. The storage of these key-value pairs is supported by Google File System (GFS) [100]. Parallelism is achieved through parallel scheduling of these distributable programs. Dryad has similar scope with MapReduce, but application dependencies can be arbitrarily constructed by explicitly specifying a DAG (directed acyclic graph). It supports data flowing along the links from one job's output to another job's input in a manner similar to a UNIX pipe.

The scheduling mechanism in MapReduce considers data locality by using the data location information provided by the GFS metadata server. Similar scheduling strategy is exploited in location-aware request distribution algorithms (LARD) for web server clusters [100], which schedule requests based on data locations as well as active connections to the servers that host the data. LARD allows dynamic creation of replicas.

Matchmaking built on the publish-subscribe model is a common approach for scheduling computationally intensive jobs. Grid computing middleware such as Condor [101] uses such a mechanism to distribute jobs. Condor matchmaking allows an agent that handles clients' job requests and resources to publish the job requirements and resource description in a form of semistructured data. The agents and resources subscribe to a matchmaker, which scans the published data and pairs jobs with resources according to the preferences and constraints specified in the data. Once a pair is formed, the matchmaker notifies the matching agent and resource. The agent and resource then establish contact and execute a job.

Dryad has a similar scheduling mechanism. Each execution of a vertex in a DAG has an execution record that contains the state of the execution and the versions of the predecessor vertices providing its input data. A vertex is placed into a scheduling queue when its input data is ready. The constraints and preferences of running the vertex are attached to its execution record (e.g., the vertex may have a list of computers on which it would like to run, or prefers to be located in a computer where a data set is stored). The scheduler then does the matchmaking by allocating the vertex to the resource. The approach has a long history and can be traced back to the Linda programming model [102] and middleware inspired by Linda, such as JavaSpaces [103] where jobs are published in a shared space generalizing the messaging queue and consumed by resources subscribed to the space.

As jobs can be dispatched to different nodes, a job execution environment typically needs the support of a distributed data management system for a job to access remote data sets and, sometimes, exchange data with other jobs. As mentioned earlier, MapReduce is supported by GFS. Dryad also has a GFS-like distributed storage system, which can break large data files into small chunks. These chunks are then distributed and replicated across the disks of nodes in the system. The communication channels between jobs are often file-based. Through a distributed storage system, a file can be passed from one job to another transparently. Accessing and exchanging data with certain structures can be supported through abstractions built on top of the distributed storage system. Google BigTable [104] and Amazon Dynamo [105] are two examples.

The data abstraction in BigTable described in Section 6.3 is a multidimensional sorted map and in Dynamo is key-value pair. With these abstractions, a job can access multidimensional data or key-value pairs from any node in the system. As a consequence, a job execution environment is very powerful nowadays. It is capable of running not only computational jobs, but also a variety of data-intensive jobs. However, as maintaining data consistency comes with a high cost in a distributed environment, such a job execution environment has limitations for running applications with strong consistency requirements. At the time of this writing, BigTable and Dynamo only support relaxed consistency.

In a large-scale distributed system where applications are dynamically placed onto any computers to run, the capability of collecting information regarding the application and resource status becomes crucial for the system to achieve efficiency, detect failures or risks, and track system states. A contemporary distributed system often has a complex monitoring subsystem. Astrolabe [106], used in Amazon, is such an example. Astrolabe monitors the state of a collection of distributed resources and is capable of continuously computing summaries of the data in the system through aggregation. Its aggregation mechanism is driven by SQL queries (e.g., the query *SELECT MIN(load) AS load* returns the minimal load in the system).

Aggregating such information dynamically from a large number of nodes is challenging. Traditional monitoring systems in clusters do not scale beyond a few dozen nodes [106]. Astrolabe adopts a decentralized approach to achieve scalability. It runs an agent on each node. The agents talk with one another using a gossip protocol. Information managed by Astrolabe is organized hierarchically. Each agent maintains a data structure that contains a subset of the hierarchy. The local information is updated directly in an agent. An agent that needs the information of an internal node of the hierarchy can aggregate the information of the children in the hierarchy from agents that are responsible for them. Information of sibling nodes in the hierarchy can be obtained from relevant agents through the gossip protocol. The gossip protocol is simple: Periodically, each agent randomly selects some other agent and exchanges information with it. The information exchanged is the least common ancestor of the two agents.

Through this mechanism, Astrolabe is capable of taking snapshots of the distributed state. The state is aggregated dynamically without a central agent storing all the information. However, the aggregated information is replicated among all the agents involved in the process and it raises consistency issues. Two users retrieving the same data attribute may get different results. Astrolabe only supports a relaxed consistency called eventual consistency, that is, given an aggregate attribute *X* that depends on some other attribute *Y*, Astrolabe guarantees that an update made to *Y* is eventually reflected in *X*. Such a monitoring system is important to a job execution environment, particularly when the underlying system scales.

5.5 WORKFLOW IN SERVICE-ORIENTED ARCHITECTURES

In Section 5.1, we described services as the basic unit for constructing distributed systems. However a "real system" consists of multiple interacting (generalized) services as shown in Figure 5.14 and introduced in Section 5.1.2.1. In particular, both a simple sensor (perhaps only an output data stream) and a complete grid (a collection of services with multiple input and output message ports) were shown in Figure 1.22 as example collections of services. Thus, the prototypical complete system could be a "grid of services," but we also talk about a "grid of grids" or even a "grid of clouds."

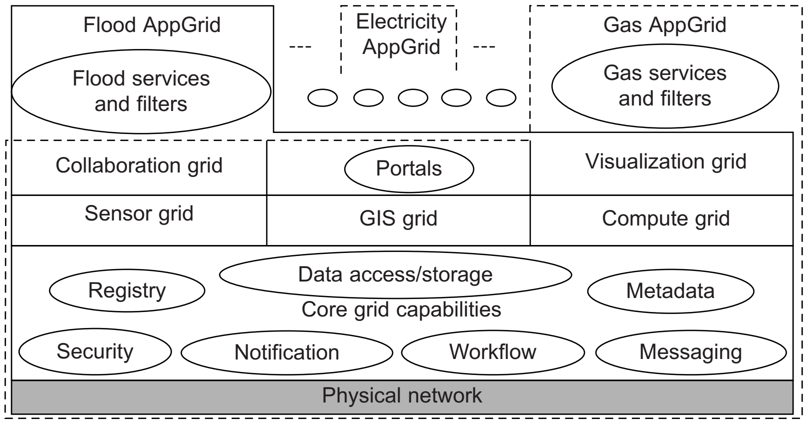

FIGURE 5.14

The concept of the grid of grids of services.

In Figure 5.14, we show multiple application grids in different areas of “critical infrastructure.” Component grids (subgrids) are invoked for collaboration, visualization, sensor fusion, computing, and GIS applications. This builds on core grid capabilities in areas covered in Sections 5.2, 5.3, and 5.4. By adding application-specific services, one builds a distributed system to support the study of flood, gas, or electrical infrastructure. Workflow is used to integrate component grids and services. Note that this discussion is phrased in terms of grids, but is equally applicable to clouds. We will discuss grid computing in more detail in Chapter 7.

5.5.1 Basic Workflow Concepts

In Section 5.2, we described approaches for managing the messages between services and component grids, but here we focus on workflow which is the approach to “programming the interaction between services.” As well as saying that workflow describes “programming the web or grid,” one can also use terms such as “software coordination,” “service orchestration,” “service or process coordination,” “service conversation,” “web or grid scripting,” “application integration,” or “software bus.”

It is an area of active research with different approaches emphasizing control flow, scheduling, and/or data flow. Some recent reviews of workflow are [15,107–110]. Note that workflow implies a two-level programming model for distributed systems. The basic services are programmed in traditional languages—C, C++, FORTRAN, Java, and Python—while workflow describes the coarser-grained programming of services interacting with one another. Each service is programmed in traditional languages while their interaction is described by workflow.

Note that the multilevel programming concept is familiar from elementary shell programming where pipes are often used to link executing programs. In fact, scripting (as in shell scripts) is one popular approach to workflow with distributed programming constructs replacing the familiar

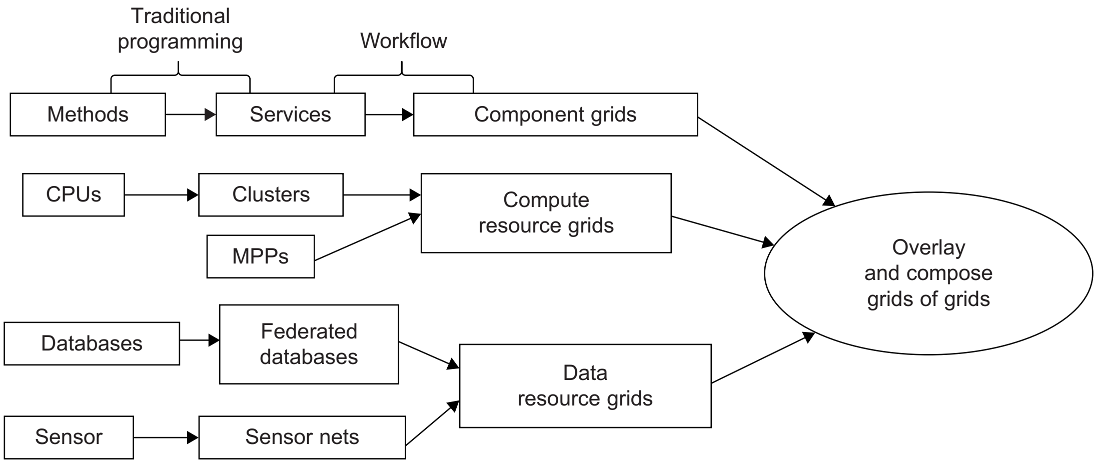

FIGURE 5.15

Hierarchical computing, data, and programming abstraction.

UNIX primitives. For example, TCP channels or publish-subscribe messaging replaces pipes. In Figure 5.15, we show that multilevel architectures are very common—not just in programming as in workflow, but also in computing, databases, and sensors.

The concept of workflow was first developed by the Workflow Management Coalition [111] which has existed for almost 20 years and generated standard reference models, documents, and a substantial industry of tools and workflow management support products. However, this coalition is largely concerned with business process management, with steps in a workflow often involving human and not computer steps. For example, Allen defined business workflow as the automation of a business process, in whole or in part, during which documents, information, or tasks are passed from one participant to another for action, according to a set of procedural rules [112]. Thus, workflow in this chapter addresses very different issues from those described by the Workflow Management Coalition.

We can identify the development of workflow concepts and systems as a major achievement of the grid and distributed system community. Note that services are, of course, equally critical, but the essence of that idea came from business systems. Historically, workflow concepts evolved from distributed programming using systems such as Linda [113], HeNCE [114], AVS [115], and Khoros [116] as well as complex Shell (Perl) scripts. Petri Nets also can be considered one of the early ideas important for workflow. Although several good workflow systems have been built on Petri Nets [117–119], the most popular systems today are not based on Petri Nets. There are important commercial areas using environments very similar to scientific workflow. These include systems to analyze experimental (laboratory) data, termed Laboratory Information Systems or LIMS as exemplified by [120,121] or the general resource [122]. The chemical informatics and business intelligence areas have several workflow-like systems including InforSense [123] and Pipeline Pilot [124].

5.5.2 Workflow Standards

As with other web service-related concepts, there was substantial standards activity among OASIS, OMG, and W3C, often with overlapping goals, as summarized in Table 5.9. This work largely stemmed from the 2000 to 2005 when standards were viewed as essential to enable the web service dream that interoperability was achieved by complete specification of service features. Recently, there has been a realization that this goal produced heavyweight architectures where the tooling could not keep up with the support of the many standards. Today we see greater emphasis on lightweight systems where interoperability is achieved by ad hoc transformations where necessary. Another problem of the standardization work was that it largely preceded the deployment of systems, and so one found premature standardization that missed key points. This background explains the many unfinished standard activities in Table 5.9.

The successful activities have a business process flavor, and for scientific workflow, the most relevant standard is BPEL [125–128], which was based on the earlier proposals WSFL and XLANG. Note that XML is not well suited to specifying programming constructs; although XML can express data structures well, it is possible but not natural to express loops and conditionals that are essential to any language and the control of a workflow. It may turn out that expressing workflow in a modern scripting language is preferable to XML-based standards.

Table 5.9 Workflow Standards, Links, and Status

Standard	Link	Status
BPEL Business Process Execution Language for Web Services (OASIS) V 2.0	http://docs.oasis-open.org/wsbpel/2.0/wsbpel-v2.0.html; http://en.wikipedia.org/wiki/BPEL	April 2007
WS-CDL Web Service Choreography Description Language (W3C)	http://www.w3.org/TR/ws-cdl-10/	November 2005, not final
WSCI Web Service Choreography Interface V 1.0 (W3C)	http://www.w3.org/TR/wsci/	August 2002, note only
WSCL Web Services Conversation Language (W3C)	http://www.w3.org/TR/wscl10/	March 2002, note only
WSFL Web Services Flow Language	http://www.ibm.com/developerworks/webservices/library/ws-wsfl2/	Replaced by BPEL
XLANG Web Services for Business Process Design (Microsoft)	http://xml.coverpages.org/XLANG-C-200106.html	June 2001, replaced by BPEL
WS-CAF Web Services Composite Application Framework including **WS-CTX, WS-CF,** and **WS-TXM**	http://en.wikipedia.org/wiki/WS-CAF	Unfinished
WS-CTX Web Services Context (OASIS Web Services Composite Application Framework TC)	http://docs.oasis-open.org/ws-caf/ws-context/v1.0/OS/wsctx.html	April 2007
WS-Coordination Web Services Coordination (BEA, IBM, Microsoft at OASIS)	http://docs.oasis-open.org/ws-tx/wscoor/2006/06	February 2009
WS-AtomicTransaction Web Services Atomic Transaction (BEA, IBM, Microsoft at OASIS)	http://docs.oasis-open.org/ws-tx/wsat/2006/06	February 2009
WS-BusinessActivity Framework (BEA, IBM, Microsoft at OASIS)	http://docs.oasis-open.org/ws-tx/wsba/2006/06	February 2009
BPMN Business Process Modeling Notation (Object Management Group, OMG)	http://en.wikipedia.org/wiki/BPMN; http://www.bpmn.org/	Active
BPSS Business Process Specification Schema (OASIS)	http://www.ebxml.org/; http://www.ebxml.org/specs/ebBPSS.pdf	May 2001
BTP Business Transaction Protocol (OASIS)	http://www.oasis-open.org/committees/download.php/12449/business_transaction-btp-1.1-spec-cd-01.doc	Unfinished

5.5.3 Workflow Architecture and Specification

Most workflow systems have two key components corresponding to the language and runtime components of any programming environment. We can call these components the workflow specification and workflow execution engine. They are linked by interfaces which could be specified by documents using, for example, the BPEL standard introduced earlier.

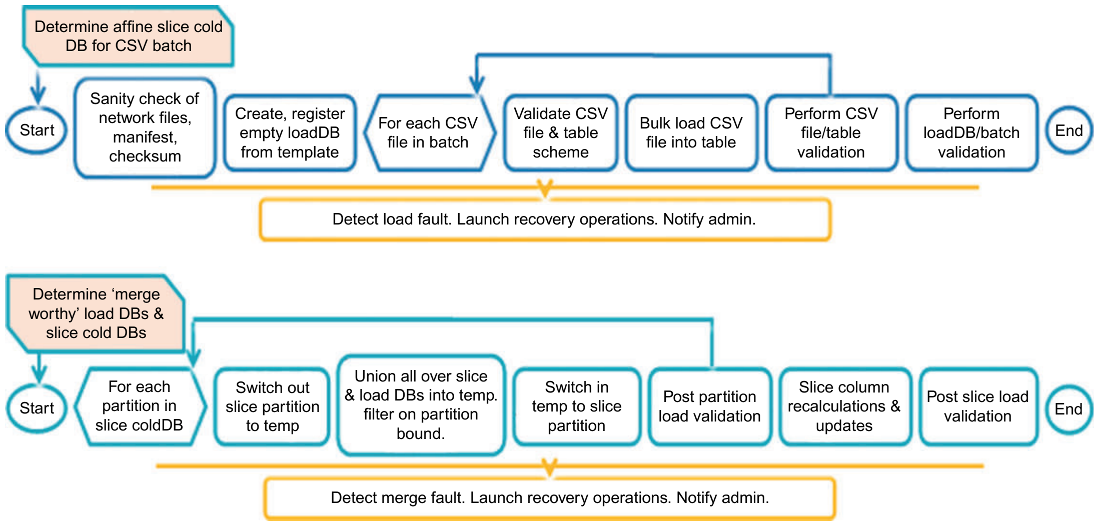
Determine affine slice cold DB for CSV batch
Start
Sanity check of network files, manifest, checksum
Create, register empty loadDB from template
For each CSV file in batch
Validate CSV file & table scheme
Bulk load CSV file into table
Perform CSV file/table validation
Perform loadDB/batch validation
End
Detect load fault. Launch recovery operations. Notify admin.
Determine 'merge worthy' load DBs & slice cold DBs
Start
For each partition in slice coldDB
Switch out slice partition to temp
Union all over slice & load DBs into temp. filter on partition bound.
Switch in temp to slice partition
Post partition load validation
Slice column recalculations & updates
Post slice load validation
End
Detect merge fault. Launch recovery operations. Notify admin.

FIGURE 5.16

Two typical (Load and Merge) workflows from the Pan-STARRS astronomy data processing area.

(*Courtesy of Barga, et al. [129]*)

5.5.3.1 *Workflow Specification*

Scripting-based workflow systems can specify workflow in traditional language syntax similar to Python, JavaScript, or Perl. One can also directly specify the (XML) interface document driving the execution engine, although that is pretty low-level. However, most workflow systems use a graphical interface as illustrated in Figure 5.16. These capture scenarios illustrated by those of following examples.

Example 5.10 Pan-STARRS Workflow

Each step in Figure 5.16 is a "large" activity (i.e., a service) illustrating that workflow is a programming system at a very different granularity from familiar languages such as C++. Note that scripting languages are often used for specifying coarse-grained operations, which "explains" why scripting is a popular approach to specifying workflows. However, a key feature of workflows involves a "few" steps, whereas a full program for a service is often thousands to perhaps millions of lines of code. This observation supports the use of a visual interface so that functional specifications are directly mapped into the programming model. Two visual interfaces are Microsoft's Trident system [130,131] and Taverna, originally developed at the University of Manchester [18,132].

Figure 5.17 shows a typical workflow specified by the Taverna system [132]. There are menus specifying components (activities, services) that are chosen by the user and represented by "boxes" in the composition window. The user specifies the workflow logic through the linkage of the "boxes." Each box has substantial metadata that can be specified and inspected by the user. The system allows specification of

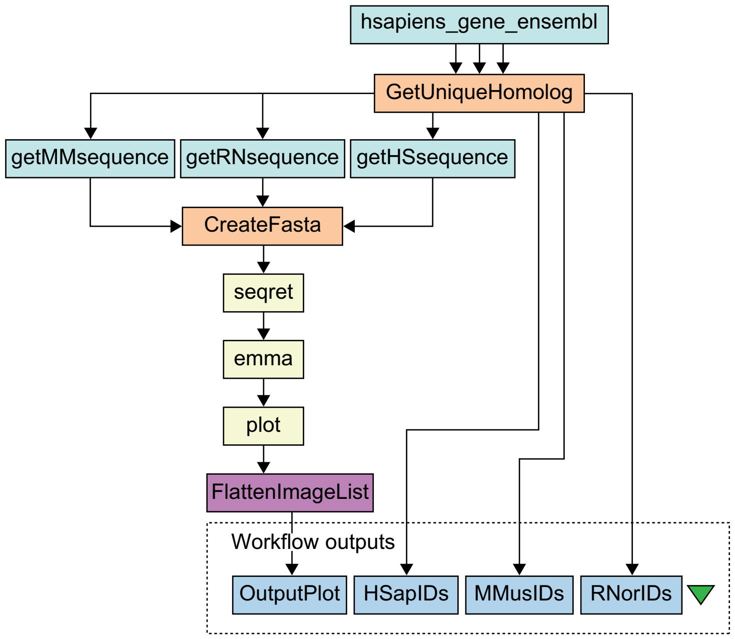

FIGURE 5.17

The workflow in the Taverna system.

(*Courtesy of C. Goble, 2008 [132]*)

nontrivial control constructs such as looping over regions of workflow and conditional decisions. Note that each box can be a sequential or parallel component.

The need for the above operations is seen in the tornado tracking workflow [131] of LEAD II [133] which involves data processing services linked to traditional (MPI) parallel weather simulations. The concurrency expressed in workflow is often termed "functional" parallelism distinguishing it from "data" parallelism. Typically functional parallelism (corresponding to different services in application) is absolute and is determined in size and nature by the problem. On the other hand, data parallelism is achieved by dividing a large data set into parts. The degree of data parallelism is determined by the number of available cores. Different parts correspond to using a parallel computing technology (MPI, threading, MapReduce) to implement the services.

5.5.4 Workflow Execution Engine

There are many different workflow systems. Workflow does not have strong performance constraints. We mentioned in the previous section that the typically large execution times of nodes made overhead less important than, say, for MPI. This same feature typically allows workflow to be executed in a distributed fashion—the network latency of long communication hops is often not important. The classification in [139] is of workflows used in domains from meteorology and ocean modeling, bioinformatics and biomedical workflows, astronomy and neutron science. These are examined according to their size, resource use, graph pattern, data pattern, and usage scenario. BPEL specifies the control and not the data flow of a workflow. Of course, the control structure implies the data flow structure for a given set of nodes.

Figure 5.18 shows a *pipeline* as seen for the astronomy workflows of Figure 5.16. A more general workflow structure is that of the DAG, which is a collection of vertices and directed edges, each edge connecting one vertex to another such that there are no cycles. That is, there is no way to start at some vertex *V* and follow a sequence of edges that eventually loops back to that vertex *V* again. In spite of sophisticated specialized workflow systems, scripting using traditional languages and toolkits is perhaps the dominant technique used to build workflows. Often this is done in an informal fashion using any environment with distributed computing (Internet) support; PHP may be the most popular environment for building mashups, but Python and JavaScript are also well used. Figure 5.18 shows a simple *cyclic graph*. Dagman [159] used in Condor [160] a sophisticated DAG processing engine. This leads to a class of workflow systems such as Pegasus [146] aimed at scheduling the nodes of DAG-based workflows. Karajan [161] and Ant [162] can also easily represent DAGs. Note that most sophisticated workflow systems support hierarchical specifications—namely the nodes of a workflow can be either services or collections of services (subworkflows). This is consistent with the Grid of Grids concept.

The issues that need to be addressed by a workflow engine are well understood in the active research community but are not well described in an integrated fashion [15,107–110]. Clearly, many and perhaps all of the execution issues discussed in this book for distributed systems, clouds, and grids are implicitly or explicitly relevant for workflow execution engines. Figure 5.19 specifies the interaction between the constituent services or activities. One important technology choice is the mechanism for transferring information between the nodes of the graph. The simplest choice is that each node reads from and writes to disk and this allows one to treat the execution of each node as an independent job invoked when all its needed input data is available on disk.

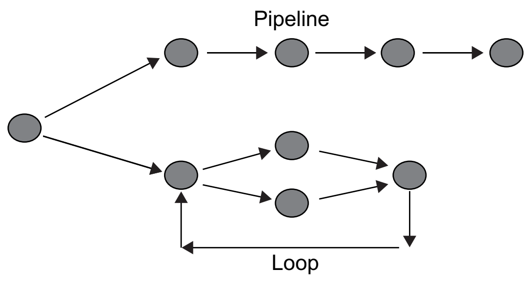

FIGURE 5.18

A workflow graph that includes subgraphs illustrating pipelines and loops.

This could seem inefficient, but we are not operating in a region for which MPI and parallel computing technologies are designed. There, low latency (microseconds) is often essential, but in a workflow, we have a different communication pattern: long-running jobs inputting and outputting substantial data sets. The cost of reading and writing is often quite acceptable and allows simpler fault tolerant implementations. Of course, one can use the messaging systems described in Section 5.2 to manage data transfer in a workflow, and in extreme cases, simple models where all communication is handled by a single central "control node."

Obviously, this latter solution could lead to poor performance that does not properly scale as workflow size increases. Distributed object technology such as CORBA can also be used for communication, as it is in the Discover Middleware [136]. In fact, there are often two communication systems in workflow environments corresponding to "control" and "data," respectively. Obviously, the control communication would usually have small messages and very different requirements from the data network. In this regard, one should mention the "proxy model" which is often used in grid architectures and workflow.

Suppose one of the nodes in Figure 5.18 corresponds to the execution of a large simulation job—say, the chemistry code Amber or Gaussian. Then one could consider the node as the Amber code with links corresponding directly to data used by this code. However, this is not normal. Rather, the node is a service that contains the metadata and mechanisms to invoke Amber on a (remote) machine and also to determine when it is complete. Then the information flowing between proxy nodes is all essentially control information. This proxy model can also be considered an agent framework [137].

5.5.5 Scripting Workflow System Swift

Swift is a parallel scripting language in which application programs are represented as functions and variables can be mapped to files. Structure and array abstractions are used to manipulate sets of files in parallel. Swift has a functional data-flow-based execution model in which all statements are implicitly parallel. Parallel looping constructs explicitly specify large-scale parallel processing. Figure 5.19 shows the architecture of the Swift workflow system.

Example 5.11 Bioinformatics Workflow in Swift

The Swift command, a Java application, runs on any user-accessible computer (e.g., a workstation or login server). It compiles and executes Swift scripts, coordinates remote data transfers, and executes applications on local and distributed parallel resources. The application of Swift to bioinformatics is illustrated in Figure 5.20. This problem is perfect for MapReduce. Based on Swift's implicit data flow model, the *foreach* loop starts automatically when the members of the array splitseqs become available.

The Swift system processes a set of protein sequences of unknown function through the BLAST search program, matching each sequence against proteins of known functions in the standard database "NR." Given, for example, 100 processors, the script can execute 100 invocations of BLAST in parallel to rapidly search the NR database. The purpose is to search for clues to the nature of the unknown proteins.

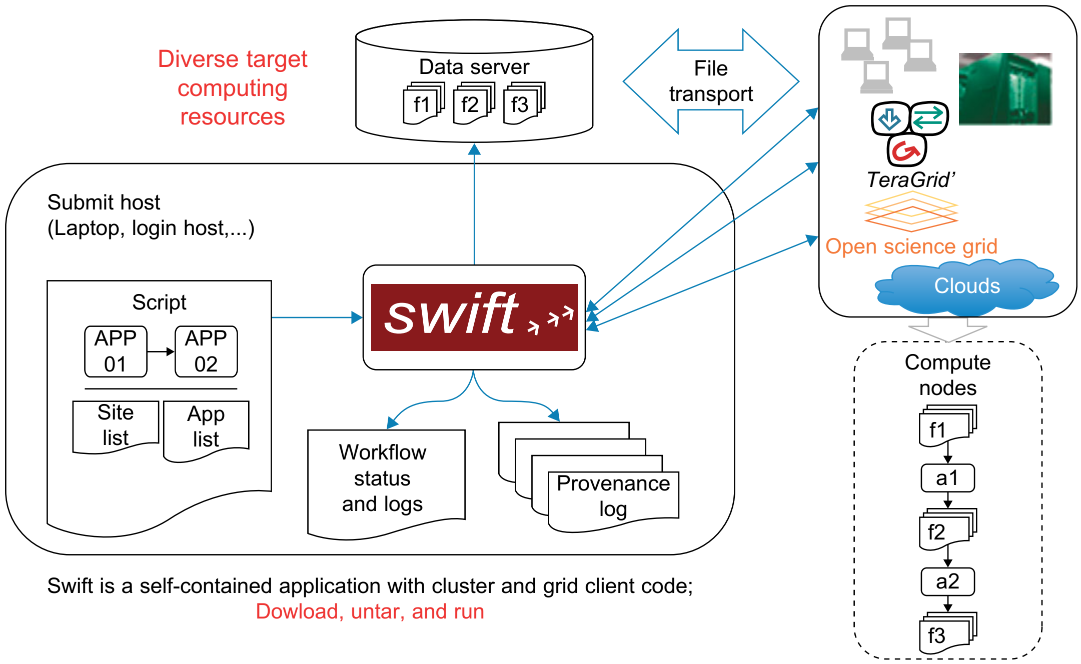

FIGURE 5.19

Swift workflow system architecture.

(*Courtesy of www.ci.uchicago.edu/swift/ [138]*)

5.6 BIBLIOGRAPHIC NOTES AND HOMEWORK PROBLEMS

We recommend the book [164] on key WS-* web service technologies. The book [23] by Alonso provides detailed coverage of services and SOA, while Fox and Gannon review in detail services used in many grid applications [15,108]. [29] and [165] discuss issues differentiating web and grid services. The important RESTful approaches are described in [8] or the original thesis in [2]. Wikipedia is often a good source of information with strong coverage of messaging, as covered in Section 5.2.

Messaging is particularly well served with quality open source systems discussed in that section. There are several good collections of papers on portals and gateways [43,44]. The semantic web is well covered with [166]. The semantic grid is covered in [167]. iRods [50] is an important grid data system while the contribution of industry to this area is very important with many seminal papers from Google and Microsoft: for example, MapReduce [98], Dryad [99], the Google File System [100], and Fusion Tables [168]. Workflow is well covered in several reviews [15,107, 108,110].

Important workflow systems based on data flow technologies are the Kepler [140–142] and Triana [143–145] projects; Kepler is still actively being developed. Pegasus is an active system

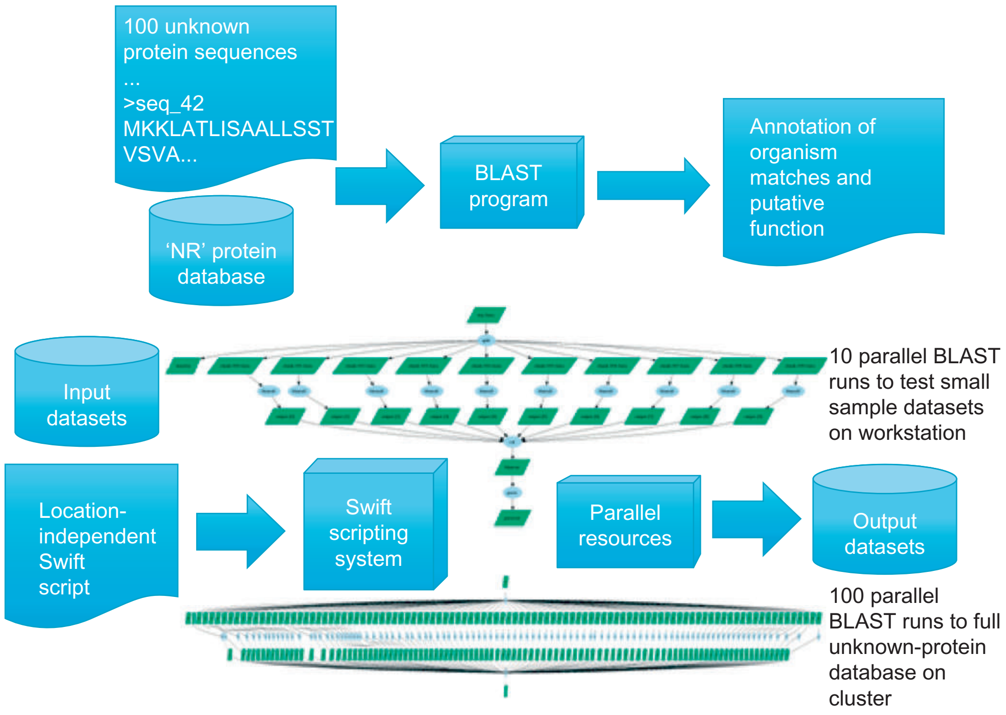
100 unknown protein sequences ... >seq_42 MKKLATLISAALLSST VSVA...
'NR' protein database
BLAST program
Annotation of organism matches and putative function
Input datasets
10 parallel BLAST runs to test small sample datasets on workstation
Location-independent Swift script
Swift scripting system
Parallel resources
Output datasets
100 parallel BLAST runs to full unknown-protein database on cluster

FIGURE 5.20

Application of the Swift to bioinformatics.

(*Courtesy of Wikipedia, http://en.wikipedia.org/wiki/Many-task_computing, [154]*)

implementing the scheduling style of workflow [146]. Taverna [18,147] from the myGrid project [148] is very popular in the bioinformatics community and substantial effort has been made by the UK OMII effort [149] into making the system robust. An innovative extension of this project is the myExperiment scientific social networking site [150], which enables sharing of workflows. Other key aspects of workflow are security [163] and fault tolerance [110].

The HPSearch project [50,151,152] at Indiana University supported workflow efficiently from JavaScript focusing on control of the messaging system, NaradaBrokering [39], used to control streaming workflows (i.e., those where data is streamed between nodes of workflow and not written to disk). There are a few other workflow systems aimed at streaming data [153]. The new scripting environment, Swift [138] supports the Many Task programming model where innumerable jobs need to be executed [154].

There are very many approaches to workflow which are largely successful in prototype one-off situations. However, experience has found that most are not really robust enough for production use outside the development team. This observation motivated Microsoft to put its Trident workflow environment [129,130] into open source for science. Trident is built on the commercial quality Windows Workflow Foundation. Note that in most cases one would use Trident in a "proxy" fashion with workflow agents running on Windows controlling remote components that largely run on Linux machines [131,133]. Some workflow systems are built around the data flow concept with this being the original model [115,116,155] and the direct opposite of the proxy concept.

Today's mashups are typically data flow systems with the interaction scripted in languages such as JavaScript or PHP. ProgrammableWeb.com has, for example, more than 2,000 APIs and 5,000 mashups recorded as of September 2010 [84]. Commercial mashup approaches include Yahoo! Pipes (http://pipes.yahoo.com/pipes/), Microsoft Popfly (http://en.wikipedia.org/wiki/Microsoft_Popfly), and IBM's sMash enhanced BPEL Web 2.0 [156] workflow environment. There is some disagreement as to whether MapReduce is a workflow environment (http://yahoo.github.com/oozie/) as it and workflow can execute similar problems. Other workflow approaches extend the "remote method invocation" model. This model underlies the *Common Component Architecture* (CCA) [157,158].

Acknowledgements

This chapter was coauthored by Geoffrey Fox of Indiana University and Albert Zomaya of Sydney University, jointly. Their work was assisted by Ali Javadzadeh Boloori and Chen Wang of the Univ. of Sydney; by Shrideep Pallickara, Marlon Pierce, and Suresh Marru of Indiana Univ.; by Michael McLennan, George B. Adams III, and Gerhard Klimeck of Purdue Univ. and by Michael Wilde of the Univ. of Chicago. Kai Hwang of USC has edited the final manuscript of this chapter.

References

[1] D. Booth, H. Haas, F. McCabe, et al., Working Group Note 11: Web Services Architecture, www.w3.org/TR/2004/NOTE-ws-arch-20040211/ (accessed 18.10.10).
[2] R. Fielding, Architectural Styles and the Design of Network-Based Software Architectures, University of California at Irvine, 2000, p. 162, http://portal.acm.org/citation.cfm?id=932295.
[3] M. Hadley, Web Application Description Language (WADL), W3C Member Submission, www.w3.org/Submission/wadl/, 2009 (accessed 18.10.10).
[4] Restlet, the leading RESTful web framework for Java, www.restlet.org/ (accessed 18.10.10).
[5] JSR 311 – JAX-RS: Java API for RESTful Web Services, https://jsr311.dev.java.net/ (accessed 18.10.10).
[6] Jersey open source, production quality, JAX-RS (JSR 311) reference implementation for building RESTful web services, https://jersey.dev.java.net/ (accessed 18.10.10).
[7] D. Winer, The XML-RPC specification, www.xmlrpc.com/, 1999 (accessed 18.10.10).
[8] L. Richardson, S. Ruby, RESTful Web Services, O'Reilly, 2007.
[9] A. Nadalin, C. Kaler, R. Monzillo, P. Hallam-Baker, Web services security: SOAP message security 1.1 (WS-Security 2004), OASIS Standard Specification, http://docs.oasis-open.org/wss/v1.1/wss-v1.1-spec-os-SOAPMessageSecurity.pdf, 2006 (accessed 18.10.10).
[10] A. Andrieux, K. Czajkowski, A. Dan, et al., Web services agreement specification (WS-Agreement), OGF Documents, GFD.107, www.ogf.org/documents/GFD.107.pdf, 2007 (accessed 18.10.2010).
[11] D. Davis, A. Karmarkar, G. Pilz, S. Winkler, Ü. Yalçinalp, Web services reliable messaging (WS-ReliableMessaging), OASIS Standard, http://docs.oasis-open.org/ws-rx/wsrm/200702, 2009 (accessed 18.10.2010).

[12] M. Little, A. Wilkinson, Web services atomic transaction (WS-AtomicTransaction) Version 1.2, OASIS Standard, http://docs.oasis-open.org/ws-tx/wstx-wsat-1.2-spec-os.pdf, 2009.

[13] M. Feingold, R. Jeyaraman, Web services coordination (WS-Coordination) Version 1.2, OASIS Standard, http://docs.oasis-open.org/ws-tx/wstx-wscoor-1.2-spec-os.pdf, 2009 (accessed 18.10.2010).

[14] K. Chiu, M. Govindaraju, R. Bramley, Investigating the limits of SOAP performance for scientific computing, in: 11th IEEE International Symposium on High Performance Distributed Computing, 2002, pp. 246–254.

[15] D. Gannon, G. Fox, Workflow in grid systems, Editorial of special issue of Concurrency & Computation: Practice & Experience, based on GGF10 Berlin meeting, Vol. 18, No. 10, 2006, pp. 1009–1019, doi: http://dx.doi.org/10.1002/cpe.v18:10 and http://grids.ucs.indiana.edu/ptliupages/publications/Workflow-overview.pdf.

[16] JBPM Flexible business process management (BPM) suite, www.jboss.org/jbpm (accessed 18.10.10).

[17] JBoss enterprise middleware, www.jboss.org/ (accessed 18.10.10).

[18] Taverna workflow management system, www.taverna.org.uk/ (accessed 18.10.10).

[19] R.V. Englen, K. Gallivan, The gSOAP toolkit for web services and peer-to-peer computing networks, in: 2nd IEEE/ACM International Symposium on Cluster Computing and The Grid (CCGRID '02), 2002.

[20] R. Salz, ZSI: The zolera soap infrastructure, http://pywebsvcs.sourceforge.net/zsi.html, 2005 (accessed 18.10.10).

[21] J. Edwards, 3-Tier server/client at work, first ed., John Wiley & Sons, 1999.

[22] B. Sun, A multi-tier architecture for building RESTful web services, IBM Developer Works 2009, www.-ibm.com/developerworks/web/library/wa-aj-multitier/index.html, 2009 (accessed 18.10.2010).

[23] G. Alonso, F. Casati, H. Kuno, V. Machiraju, Web Services: Concepts, Architectures and Applications (Data-Centric Systems and Applications), Springer Verlag, 2010.

[24] I. Foster, H. Kishimoto, A. Savva, et al., The open grid services architecture version 1.5, Open Grid Forum, GFD.80, www.ogf.org/documents/GFD.80.pdf, 2006.

[25] I. Foster, S. Tuecke, C. Kesselman, The philosophy of the grid, in: 1st International Symposium on Cluster Computing and the Grid (CCGRID0), IEEE Computer Society, 2001.

[26] S. Tuecke, K. Czajkowski, I. Foster, et al., Grid Services Infrastructure (OGSI) Version 1.0. Global Grid Forum Proposed Recommendation, GFD15, www.ggf.org/documents/GFD.15.pdf, 2003 (accessed 18.10.10).

[27] S. Graham, A. Karmarkar, J. Mischkinsky, I. Robinson, I. Sedukhin, Web Services Resource 1.2 (WS-Resource) WSRF, OASIS Standard, http://docs.oasis-open.org/wsrf/wsrf-ws_resource-1.2-spec-os.pdf, 2006 (accessed 18.10.10).

[28] M. Gudgin, M. Hadley, T. Rogers, Web Services Addressing 1.0 – Core, W3C Recommendation, 9 May 2006.

[29] G. Fox, D. Gannon, A Survey of the Role and Use of Web Services and Service Oriented Architectures in Scientific/Technical Grids, http://grids.ucs.indiana.edu/ptliupages/publications/ReviewofServicesandWorkflow-IU-Aug2006B.pdf, 2006 (accessed 16.10.10).

[30] Supported version of Mule ESB, www.mulesoft.com/.

[31] Open source version of Mule ESB, www.mulesoft.org/.

[32] IBM's original network software MQSeries rebranded WebSphereMQ in 2002, http://en.wikipedia.org/wiki/IBM_WebSphere_MQ.

[33] WebSphereMQ IBM network software, http://en.wikipedia.org/wiki/IBM_WebSphere_MQ.

[34] H. Shen, Content-based publish/subscribe systems, in: X. Shen, et al., (Eds.), Handbook of Peer-to-Peer Networking, Springer Science+Business Media, LLC, 2010, pp. 1333–1366.

[35] Java Message Service JMS API, www.oracle.com/technetwork/java/index-jsp-142945.html and http://en.wikipedia.org/wiki/Java_Message_Service.

[36] AQMP Open standard for messaging middleware, www.amqp.org/confluence/display/AMQP/Advanced+Message+Queuing+Protocol.
[37] Mule MQ open source enterprise-class Java Message Service (JMS) implementation, www.mulesoft.org/documentation/display/MQ/Home.
[38] Apache ActiveMQ open source messaging system, http://activemq.apache.org/.
[39] NaradaBrokering open source content distribution infrastructure, www.naradabrokering.org/.
[40] RabbitMQ open source Enterprise Messaging System, www.rabbitmq.com/.
[41] Amazon Simple Queue Service (Amazon SQS), http://aws.amazon.com/sqs/.
[42] Microsoft Azure Queues, http://msdn.microsoft.com/en-us/windowsazure/ff635854.aspx.
[43] N. Wilkins-Diehr, Special issue: Science gateways – common community interfaces to grid resources, Concurr. Comput. Pract. Exper. 19 (6) (2007) 743–749.
[44] N. Wilkins-Diehr, D. Gannon, G. Klimeck, S. Oster, S. Pamidighantam, TeraGrid science gateways and their impact on science, Computer 41 (11) (2008).
[45] Y. Liu, S. Wang, N. Wilkins-Diehr, SimpleGrid 2.0: A learning and development toolkit for building highly usable TeraGrid science gateways, in: SC-GCE, 2009.
[46] D.A. Reed, Grids, the TeraGrid and beyond, IEEE Comput. 36 (1) (2003) 62–68.
[47] R. Pordes, D. Petravick, B. Kramer, et al., The open science grid, J. Phys. Conf. Ser. 78 (2007) 012057.
[48] I. Foster, Globus toolkit version 4: software for service-oriented systems, J. Comput. Sci. Technol. 21 (4) (2006) 513–520.
[49] D. Thain, T. Tannenbaum, M. Livny, Distributed computing in practice: the condor experience, Concurr. Pract. Exper. 17 (2–4) (2005) 323–356.
[50] IRODS: Data Grids, Digital Libraries, Persistent Archives, and Real-time Data Systems, https://www.irods.org (accessed 29.08.10).
[51] J. Basney, M. Humphrey, V. Welch, The MyProxy online credential repository, Softw. Pract. Exp. 35 (9) (2005) 801–816.
[52] J.V. Welch, J. Basney, D. Marcusiu, N. Wilkins-Diehr, A AAAA model to support science gateways with community accounts, Concurr. Comput. Pract. Exper., (2006) 893–904.
[53] L. Liming, et al., TeraGrid's integrated information service, in: 5th Grid Computing Environments Workshop, ACM, 2009.
[54] J. Kim, P.V. Sudhakar, Computational Chemistry Grid, a Production Cyber-environment through Distributed Computing: Recent Enhancements and Application for DFT Calculation of Amide I Spectra of Amyloid-Fibril. PRAGMA13, Urbana, IL, 2007.
[55] R. Dooley, K. Milfeld, C. Guiang, S. Pamidighantam, G. Allen, From proposal to production: lessons learned developing the computational chemistry grid cyberinfrastructure, J. Grid Comput. 4 (2) (2006) 195–208.
[56] GridChem-Related Scientific Publications, https://www.gridchem.org/papers.
[57] B. Demeler, UltraScan: A comprehensive data analysis software package for analytical ultracentrifugation experiments, in: Analytical Ultracentrifugation:Techniques and Methods, 2005, pp. 210–229.
[58] UltraScan Gateway, http://uslims.uthscsa.edu/.
[59] UltraScan publications database, www.ultrascan.uthscsa.edu/search-refs.html.
[60] E.H. Brookes, R.V. Boppana, B. Demeler, Biology – Computing large sparse multivariate optimization problems with an application in biophysics, in: ACM/IEEE Conference on Supercomputing (SC2006), Tampa FL, 2006.
[61] E.H. Brookes, B. Demeler, Parsimonious regularization using genetic algorithms applied to the analysis of analytical ultracentrifugation experiments, in: 9th Annual Conference on Genetic and Evolutionary Computation (GECCO), London, 2007, pp. 361–368.
[62] G. Klimeck, M. McLennan, S.P. Brophy, et al., nanoHUB.org: Advancing education and research in nanotechnology, Comput. Sci. Eng. 10 (5) (2008) 17–23.

[63] A. Strachan, G. Klimeck, M.S. Lundstrom, Cyber-enabled simulations in nanoscale science and engineering, Comput. Sci. Eng. 12 (2010) 12–17.
[64] H. Abelson, The creation of OpenCourseWare at MIT, J. Sci. Educ. Technol. 17 (2) (2007) 164–174.
[64a] T. J. Hacker, R. Eigenmann, S. Bagchi, A. Irfanoglu, S. Pujol, A. Catlin, Ellen Rathje, The NEEShub cyberinfrastructure for earthquake engineering, computing, in Science and Engineering, 13 (4) (2011) 67–78, doi:10.1109/MCSE.2011.70.
[65] T. Richardson, Q. Stafford-Fraser, K.R. Wood, A. Hopper, Virtual network computing, IEEE Internet Comput. 2 (1) (1998) 33–38.
[66] P.M. Smith, T.J. Hacker, C.X. Song, Implementing an industrial-strength academic cyberinfrastructure at Purdue University, in: IEEE International Parallel and Distributed Processing Symposium (IPDPS), 2008.
[67] W. Qiao, M. McLennan, R. Kennell, D. Ebert, G. Klimeck, Hub-based simulation and graphics hardware accelerated visualization for nanotechnology applications, IEEE Trans. Vis. Comput. Graph. 12 (2006) 1061–1068.
[68] G. Klimeck, M. Luisier, Atomistic modeling of realistically extended semiconductor devices with NEMO/OMEN, IEEE Comput. Sci. Eng. 12 (2010) 28–35.
[69] B.P. Haley, G. Klimeck, M. Luisier, et al., Computational nanoelectronics research and education at nanoHUB.org, J. Comput. Electron. 8 (2009) 124–131.
[70] OpenVZ web site, http://openvz.org (accessed 17.08.10).
[71] J. Alameda, M. Christie, G. Fox, et al., The open grid computing environments collaboration: portlets and services for science gateways, Concurr. Comput. Pract. Exper. 19 (6) (2007) 921–942.
[72] Open Grid Computing Environments web site, www.collab-ogce.org (accessed 18.10.10).
[73] Z. Guo, R. Singh, M.E. Pierce, Building the PolarGrid portal using Web 2.0 and OpenSocial, in: SC-GCE, 2009.
[74] T. Gunarathne, C. Herath, E. Chinthaka, S. Marru, Experience with adapting a WS-BPEL runtime for eScience workflows, in: 5th Grid Computing Environments Workshop, ACM, 2009.
[75] S. Marru, S. Perera, M. Feller, S. Martin, Reliable and Scalable Job Submission: LEAD Science Gateways Testing and Experiences with WS GRAM on TeraGrid Resources, in: TeraGrid Conference, 2008.
[76] S. Perera, S. Marru, T. Gunarathne, D. Gannon, B. Plale, Application of management frameworks to manage workflow-based systems: A case study on a large scale e-science project, in: IEEE International Conference on Web Services, 2009.
[77] S. Perera, S. Marru, C. Herath, Workflow Infrastructure for Multi-scale Science Gateways, in: TeraGird Conference, 2008.
[78] T. Andrews, F. Curbera, H. Dholakia, et al., Business process execution language for web services, version 1.1, 2003.
[79] T. Oinn, M. Addis, J. Ferris, et al., Taverna: a tool for the composition and enactment of bioinformatics workflows, Bioinformatics, (2004).
[80] E. Deelman, J. Blythe, Y. Gil, et al., Pegasus: Mapping scientific workflows onto the grid, in: Grid Computing, Springer, 2004.
[81] Apache ODE (Orchestration Director Engine) open source BPEL execution engine, http://ode.apache.org/.
[82] M. Christie, S. Marru, The LEAD Portal: a TeraGrid gateway and application service architecture, Concurr. Comput. Pract. Exper. 19 (6) (2007) 767–781.
[83] UDDI Version 3 Specification, OASIS Standard, OASIS UDDI Specifications TC – Committee Specifications, www.oasis-open.org/committees/uddi-spec/doc/tcspecs.htm#uddiv3, 2005 (accessed 18.10.10).
[84] Programmable Web site for contributed service APIs and mashups, www.programmableweb.com/ (accessed 18.10.10).
[85] J. Gregorio, B. de hOra, RFC 5023 – The Atom Publishing Protocol, IETF Request for Comments, http://tools.ietf.org/html/rfc5023, 2007 (accessed 18.10.10).
[86] L. Vargas, J. Bacon, Integrating Databases with Publish/Subscribe, in: 25th IEEE International Conference on Distributed Computing Systems Workshops (ICDCSW '05), 2005.

[87] M. Aktas, Thesis. Information Federation in Grid Information Services. Indiana University, http://grids.ucs.indiana.edu/ptliupages/publications/MehmetAktasThesis.pdf, 2007.
[88] M.S. Aktas, G.C. Fox, M. Pierce, A federated approach to information management in grids, J. Web Serv. Res. 7 (1) (2010) 65–98, http://grids.ucs.indiana.edu/ptliupages/publications/JWSR-PaperRevisedSubmission529-Proofread.pdf.
[89] M.S. Aktas, M. Pierce, High-performance hybrid information service architecture, Concurr. Comput. Pract. Exper. 22 (15) (2010) 2095–2123.
[90] E. Deelman, G. Singh, M.P. Atkinson, et al., Grid based metadata services, in: 16th International Conference on Scientific and Statistical DatabaseManagement (SSDBM '04), Santorini, Greece, 2004.
[91] C.K. Baru, The SDSC Storage Resource Broker, in: I. Press, (Ed.), CASCON '98 Conference, Toronto, 1998.
[92] G. Singh, S. Bharathi, A. Chervenak, et al., A Metadata Catalog Service for Data Intensive Applications, in: 2003 ACM/IEEE Conference on Supercomputing, Conference on High Performance Networking and Computing, ACM Press, 15–21 November 2003, p. 17.
[93] L. Chervenak, et al., Performance and Scalability of a Replica Location Service, in: 13th IEEE International Symposium on High Performance Distributed Computing (HPDC 13), IEEE Computer Society, Washington, DC, 2004, pp. 182–191.
[94] B. Koblitz, N. Santos, V. Pose, The AMGA metadata service, J. Grid Comput. 6 (1) (2007) 61–76.
[95] C.A. Goble, D. De Roure, The Semantic Grid: Myth busting and bridge building, in: 16th European Conference on Artificial Intelligence (ECAI-2004), Valencia, Spain, 2004.
[96] O. Corcho, et al., An Overview of S-OGSA: A Reference Semantic Grid Architecture, in: Journal of Web Semantics: Science, Services and Agents on the World Wide Web, 2006, pp. 102–115.
[97] MyGrid, www.mygrid.org.uk/.
[98] J. Dean, S. Ghemawat, MapReduce: Simplified Data Processing on Large Clusters, in: Sixth Symposium on Operating Systems Design and Implementation, 2004, pp. 137–150.
[99] M. Isard, M. Budiu, Y. Yu, A. Birrell, D. Fetterly, Dryad: Distributed Data-Parallel Programs from Sequential Building Blocks, in: ACM SIGOPS Operating Systems Review, ACM Press, Lisbon, Portugal, 2007.
[100] S. Ghemawat, The Google File System, in: 19th ACM Symposium on Operating System Principles, 2003, pp. 20–43.
[101] D. Thain, T. Tannenbaum, M. Livny, Distributed computing in practice: the Condor experience, Concurr. Comput. Pract. Exper. 17 (2–4) (2005) 323–356.
[102] D. Gelernter, Generative communication in Linda, in: ACM Transactions on Programming Languages and Systems, 1985, pp. 80–112.
[103] E. Freeman, S. Hupfer, K. Arnold, JavaSpaces: Principles, Patterns, and Practice, Addison-Wesley, 1999.
[104] F. Chang, et al., BigTable: A Distributed Storage System for Structured Data, in: OSDI 2006, Seattle, pp. 205–218.
[105] G. De Candia, et al., Dynamo: Amazon's highly available key-value store, in: SOSP, Stevenson, WA, pp. 205–219.
[106] R. Van Renesse, K. Birman, W. Vogels, Astrolabe: A robust and scalable technology for distributed system monitoring, management, and data mining, in: ACM Transactions on Computer Systems, 2003, pp. 164–206.
[107] J. Yu, R. Buyya, A taxonomy of workflow management systems for grid computing, in: Technical Report, GRIDS-TR-2005-1, Grid Computing and Distributed Systems Laboratory, University of Melbourne, Australia, 2005.
[108] I.J. Taylor, E. Deelman, D.B. Gannon, M. Shields, Workflows for e-Science: Scientific Workflows for Grids, Springer, 2006.
[109] Z. Zhao, A. Belloum, M. Bubak, Editorial: Special section on workflow systems and applications in e-Science, Future Generation Comp. Syst. 25 (5) (2009) 525–527, http://dx.doi.org/10.1016/j.future.2008.10.011.

[110] E. Deelman, D. Gannon, M. Shields, I. Taylor, Workflows and e-Science: an overview of workflow system features and capabilities, Future Generation Comp. Syst. 25 (5) (2009) 528–540, doi: http://dx.doi.org/10.1016/j.future.2008.06.012.
[111] Workflow Management Consortium, www.wfmc.org/.
[112] R. Allen, Workflow: An Introduction, Workflow Handbook. Workflow Management Coalition, 2001.
[113] N. Carriero, D. Gelernter, Linda in context, Commun. ACM 32 (4) (1989) 444–458.
[114] A. Beguelin, J. Dongarra, G.A. Geist, HeNCE: A User's Guide, Version 2.0, www.netlib.org/hence/hence-2.0-doc-html/hence-2.0-doc.html.
[115] C. Upson, T. Faulhaber Jr., D.H. Laidlaw, et al., The application visualization system: a computational environment for scientific visualization, IEEE Comput. Graph. Appl., (1989) 30–42.
[116] J. Rasure, S. Kubica, The Khoros application development environment, in: Khoral Research Inc., Albuquerque, New Mexico, 1992.
[117] A. Hoheisel, User tools and languages for graph-based Grid workflows, Concurr. Comput. Pract. Exper. 18 (10) (2006) 1101–1113, http://dx.doi.org/10.1002/cpe.v18:10.
[118] Z. Guan, F. Hernandez, P. Bangalore, et al., Grid-Flow: A Grid-enabled scientific workflow system with a Petri-net-based interface, Concurr. Comput. Pract. Exper. 18 (10) (2006) 1115–1140, http://dx.doi.org/10.1002/cpe.v18:10.
[119] M. Kosiedowski, K. Kurowski, C. Mazurek, J. Nabrzyski, J. Pukacki, Workflow applications in GridLab and PROGRESS projects, Concurr. Comput. Pract. Exper. 18 (10) (2006) 1141–1154, http://dx.doi.org/10.1002/cpe.v18:10.
[120] LabVIEW Laboratory Virtual Instrumentation Engineering Workbench, http://en.wikipedia.org/wiki/LabVIEW.
[121] LabSoft LIMS laboratory information management system, www.labsoftlims.com/.
[122] LIMSource Internet LIMS resource, http://limsource.com/home.html.
[123] InforSense Business Intelligence platform, hwww.inforsense.com/products/core_technology/inforsense_platform/index.html.
[124] Pipeline Pilot scientific informatics platform from Accelrys, http://accelrys.com/products/pipeline-pilot/.
[125] OASIS Web Services Business Process Execution Language Version 2.0 BPEL, http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html.
[126] F. Curbera, R. Khalaf, W.A. Nagy, S. Weerawarana, Implementing BPEL4WS: The architecture of a BPEL4WS implementation, Concurr. Comput. Pract. Exper. 18 (10) (2006) 1219–1228, http://dx.doi.org/10.1002/cpe.v18:10.
[127] ActiveBPEL Open Source workflow engine, www.activebpel.org/.
[128] F. Leyman, Choreography for the Grid: Towards fitting BPEL to the resource framework, Concurr. Comput. Pract. Exper. 18 (10) (2006) 1201–1217, http://dx.doi.org/10.1002/cpe.v18:10.
[129] R. Barga, D. Guo, J. Jackson, N. Araujo, Trident: a scientific workflow workbench, in: Tutorial eScience Conference, Indianapolis, 2008.
[130] Microsoft, Project Trident: A Scientific Workflow Workbench, http://tridentworkflow.codeplex.com/ and http://research.microsoft.com/en-us/collaboration/tools/trident.aspx.
[131] The forecast before the storm. [iSGTW International Science Grid This Week], www.isgtw.org/?pid=1002719, 2010.
[132] C. Goble, Curating services and workflows: the good, the bad and the ugly, a personal story in the small, in: European Conference on Research and Advanced Technology for Digital Libraries, 2008.
[133] Linked Environments for Atmospheric Discovery II (LEAD II), http://pti.iu.edu/d2i/leadII-home.
[134] XBaya workflow composition tool, www.collab-ogce.org/ogce/index.php/XBaya.
[135] XBaya integration with OGCE Open Grid Computing Environments Portal, www.collab-ogce.org/ogce/index.php/XBaya.

[136] V. Bhat, M. Parashar, Discover middleware substrate for integrating services on the grid, in: Proceedings of the 10th International Conference on High Performance Computing (HiPC 2003), Lecture Notes in Computer Science. Springer-Verlag, Hyderabad, India, 2003.
[137] Z. Zhao, A. Belloum, C.D. Laat, P. Adriaans, B. Hertzberger, Distributed execution of aggregated multi-domain workflows using an agent framework, in: IEEE Congress on Services (Services 2007), 2007.
[138] Open source scripting workflow supporting the many task execution paradigm, www.ci.uchicago.edu/swift/.
[139] L. Ramakrishnan, B. Plale, A multi-dimensional classification model for scientific workflow characteristics, in: 1st International Workshop on Workflow Approaches to New Data-Centric Science, Indianapolis, 2010.
[140] Kepler Open Source Scientific Workflow System, http://kepler-project.org.
[141] B. Ludäscher, I. Altintas, C. Berkley, et al., Scientific workflow management and the Kepler system, Concurr. Comput. Pract. Exper. 18 (10) (2006) 1039–1065, http://dx.doi.org/10.1002/cpe.v18:10.
[142] T. McPhillips, S. Bowers, D. Zinn, B. Ludäscher, Scientific workflow design for mere mortals, Future Generation Comp. Syst. 25 (5) (2009) 541–551, http://dx.doi.org/10.1016/j.future.2008.06.013.
[143] Triana, Triana Open Source Problem Solving Environment, www.trianacode.org/index.html (accessed 18.10.10).
[144] I. Taylor, M. Shields, I. Wang, A. Harrison, in: I. Taylor, et al., (Eds.), Workflows for e-Science, Springer, 2007, pp. 320–339.
[145] D. Churches, G. Gombas, A. Harrison, et al., Programming scientific and distributed workflow with Triana services, Concurr. Comput. Pract. Exper. 18 (10) (2006) 1021–1037, http://dx.doi.org/10.1002/cpe.v18:10.
[146] Pegasus Workflow Management System, http://pegasus.isi.edu/.
[147] T. Oinn, M. Greenwood, M. Addis, et al., Taverna: Lessons in creating a workflow environment for the life sciences, Concurr. Comput. Pract. Exper. 18 (10) (2006) 1067–1100, http://dx.doi.org/10.1002/cpe.v18:10.
[148] myGrid multi-institutional, multi-disciplinary research group focusing on the challenges of eScience, www.mygrid.org.uk/.
[149] OMII UK Software Solutions for e-Research, www.omii.ac.uk/index.jhtml.
[150] Collaborative workflow social networking site, www.myexperiment.org/.
[151] H. Gadgil, G. Fox, S. Pallickara, M. Pierce, Managing grid messaging middleware, in: Challenges of Large Applications in Distributed Environments (CLADE), 2006, pp. 83–91.
[152] HPSearch, Scripting environment for managing streaming workflow and their messaging based communication, www.hpsearch.org/, 2005 (accessed 18.10.10).
[153] C. Herath, B. Plale, Streamflow, in: 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing, 2010.
[154] Many Task Computing Paradigm, http://en.wikipedia.org/wiki/Many-task_computing.
[155] D. Bhatia, V. Burzevski, M. Camuseva, et al., WebFlow: A visual programming paradigm for web/Java based coarse grain distributed computing, Concurr. Comput. Pract. Exper. 9 (6) (1997) 555–577.
[156] IBM WebSphere sMash Web 2.0 Workflow system, IBM DeveloperWorks, www.ibm.com/developerworks/websphere/zones/smash/ (accessed 19.10.10).
[157] Common Component Architecture CCA Forum, www.cca-forum.org/.
[158] D. Gannon, S. Krishnan, L. Fang, et al., On building parallel & grid applications: component technology and distributed services, Cluster Comput. 8 (4) (2005) 271–277, http://dx.doi.org/10.1007/s10586-005-4094-2.
[159] E. Deelman, T. Kosar, C. Kesselman, M. Livny, What makes workflows work in an opportunistic environment? Concurr. Comput. Pract. Exper. 18 (10) (2006), http://dx.doi.org/10.1002/cpe.v18:10.
[160] Condor home page, www.cs.wisc.edu/condor/.
[161] Karajan parallel scripting language, http://wiki.cogkit.org/index.php/Karajan.
[162] GridAnt extension of the Apache Ant build tool residing in the Globus COG kit, www.gridworkflow.org/snips/gridworkflow/space/GridAnt.

[163] H. Chivers, J. McDermid, Refactoring service-based systems: How to avoid trusting a workflow service, Concurr. Comput. Pract. Exper. 18 (10) (2006) 1255–1275, http://dx.doi.org/10.1002/cpe.v18:10.

[164] S. Weerawarana, F. Curbera, F. Leymann, T. Storey, D.F. Ferguson. Web Services Platform Architecture: SOAP, WSDL, WS-Policy, WS-Addressing, WS-BPEL, WS-Reliable Messaging, and More, Prentice Hall, 2005.

[165] M. Atkinson, D. DeRoure, A. Dunlop, et al., Web Service Grids: An evolutionary approach, Concurr. Comput. Pract. Exper. 17 (2005) 377–389, http://dx.doi.org/10.1002/cpe.936.

[166] T. Segaran, C. Evans, J. Taylor, Programming the Semantic Web, O'Reilly, 2009.

[167] G. Fox, Data and metadata on the semantic grid, Comput. Sci. Eng. 5 (5) (2003) 76–78.

[168] H. Gonzalez, A. Halevy, C.S. Jensen, et al., Google fusion tables: Data management, integration and collaboration in the cloud. International Conference on Management of Data, in: Proceedings of the 1st ACM Symposium on Cloud Computing, ACM, Indianapolis, 2010, pp. 175–180.

HOMEWORK PROBLEMS

Problem 5.1

Discuss the strengths and drawbacks of WS-* and RESTful web services. Compare their architectural principles. Which one is the preferred mechanism for communicating with Amazon S3? Why?

Problem 5.2

Discuss advantages and shortcomings of stateless web services. How can we keep the state in REST and standard web services?

Problem 5.3

This assignment requires you to combine the queuing and publish-subscribe paradigms within a single application. Organizations A and B are two businesses that use queuing for B2B transactions. Every transaction is stored (prior to forwarding) and has a 128-bit UUID identifier associated with it. Within the organization, messages are delivered using publish-subscribe. Create five subscribers (sales, marketing, audit, packaging, and finance) within each organization; these subscribers do not have to log the messages again since there is a copy of that message already available. If the assignment is done in Java it is prescribed that the Java Message Service be used.

Problem 5.4

Develop an application that uses publish-subscribe to communicate between entities developed in different languages. Producers should be written in C++, and consumers should be in Java. Distributed components communicate with one another using the AMQP wire format.

Problem 5.5

Map the terms and abbreviations on the left to the best suited descriptions on the right.

1. _____ S-OGSA	(a) A set of specifications defining a way to describe, publish, and discover information about web services
2. _____ REST	(b) The first language developed for the Semantic web, using XML to represent information about resources on the web
3. _____ UDDI	(c) An XML vocabulary to describe RESTful web services
4. _____ MCS	(d) A family of OASIS-published specifications which provide a set of operations that web services may implement to become stateful
5. _____ WSDL	(e) A catalog service developed by the Globus Alliance providing hierarchical organization of the metadata
6. _____ RDF	(f) Composite Web 2.0 applications which combine capabilities from existing web-based applications
7. _____ WADL	(g) A software architecture for distributed hypermedia systems based on HTTP
8. _____ WSRF	(h) An expressive ontology language that extends RDF schema
9. _____ Mashups	(i) A reference architecture for semantic-based grids by extending Open Grid Services Architecture
10. _____ OWL-S	(j) An XML-based language to describe the set of operations and messages supported by a web service

Problem 5.6

Combine the two assignments 5.3 and 5.4 above, where organization A is developed in Java, while organization B is developed in C++. Use AMQP format in these communications.

Problem 5.7

Describe possible uses of enterprise buses in grid architectures.

Problem 5.8

Run NaradaBrokering or an equivalent system selected from Table 5.8 in an available cloud. Use it to connect two Android smartphones exchanging messages and pictures snapped by the phone.

Problem 5.9

Install the OGCE toolkit using instructions at the web site: www.collabogce.org/ogce/index.php/Portal_download. Then select one or more from the range of projects given at another web site: www.collab-ogce.org/ogce/index.php/Tutorials#Screen_Capture_Demos.

Problem 5.10

Compare Google App Engine, Drupal, HUBzero, and OGCE approaches to science gateways. Discuss in the context of portals you are familiar with.

Problem 5.11

What are the three main categories of information that a service registry contains? Which entities map these categories of information in UDDI?

Problem 5.12

Outline the main components of a real-life parallel or distributed job execution environment. You can choose any example platforms such as clusters in Chapter 2, clouds in Chapter 4, grids in Chapter 7, and P2P networks in Chapter 8.

Problem 5.13

Describe the role of a distributed file system in a job execution environment such as MapReduce in a large-scale cloud system.

Problem 5.14

Resource Description Framework (RDF) and the OWL web ontology language are two technologies for the Semantic web. Describe the relationship between them.

Problem 5.15

Why are grids popular in academic applications while clouds dominate commercial use? Use case studies to compare their strengths and weaknesses.

Problem 5.16

Study more details from [138,154]. Implement the Swift application shown in Figure 5.20 on an available cloud platform.

Problem 5.17

Use Swift to implement the word count problem described in MapReduce (Details of MapReduce are given in Section 6.2.2).

Problem 5.18

Use Taverna to build a workflow linking a module to extract comments on grids and clouds (or your favorite topic) from Twitter, Flickr, or Facebook. The social networking APIs can be found at www.programmableweb.com.